

1

Introduction to JavaScript and the Web

In this introductory chapter, you look at what JavaScript is, what it can do for you, and what you need in order to use it. With these foundations in place, you will see throughout the rest of the book how JavaScript can help you to create powerful web applications for your web site.

The easiest way to learn something is by actually doing it, so throughout the book you'll create a number of useful example programs using JavaScript. This process starts in this chapter, by the end of which you will have created your first piece of JavaScript code.

Introduction to JavaScript

In this section you take a brief look at what JavaScript is, where it came from, how it works, and what sorts of useful things you can do with it.

What Is JavaScript?

Having bought this book, you are probably already well aware that JavaScript is some sort of *computer language*, but what is a computer language? Put simply, a computer language is a series of instructions that tell the computer to do something. That something can be one of a wide variety of things, including displaying text, moving an image, or asking the user for information. Normally, the instructions, or what is termed *code*, are *processed* from the top line downward. This simply means that the computer looks at the code you've written, works out what action you want taken, and then takes that action. The act of processing the code is called *running* or *executing* it.

In natural English, here are instructions, or code, you might write to make a cup of instant coffee:

1. Put coffee crystals in cup.
2. Fill kettle with water.

Chapter 1: Introduction to JavaScript and the Web

3. Put kettle on to boil.
4. Has the kettle boiled? If so, then pour water into cup; otherwise, continue to wait.
5. Drink coffee.

You'd start running this code from the first line (instruction 1), and then continue to the next (instruction 2), then the next, and so on until you came to the end. This is pretty much how most computer languages work, JavaScript included. However, there are occasions when you might change the flow of execution or even skip over some code, but you'll see more of this in Chapter 3.

JavaScript is an interpreted language rather than a compiled language. What is meant by the terms *interpreted* and *compiled*?

Well, to let you in on a secret, your computer doesn't really understand JavaScript at all. It needs something to interpret the JavaScript code and convert it into something that it understands; hence it is an *interpreted language*. Computers understand only *machine code*, which is essentially a string of binary numbers (that is, a string of zeros and ones). As the browser goes through the JavaScript, it passes it to a special program called an *interpreter*, which converts the JavaScript to the machine code your computer understands. It's a bit like having a translator translate English to Spanish, for example. The important point to note is that the conversion of the JavaScript happens at the time the code is run; it has to be repeated every time this happens. JavaScript is not the only interpreted language; there are others, including VBScript.

The alternative *compiled language* is one in which the program code is converted to machine code before it's actually run, and this conversion has to be done only once. The programmer uses a compiler to convert the code that he wrote to machine code, and this machine code is run by the program's user. Compiled languages include Visual Basic and C++. Using a real-world analogy, it's a bit like having a Spanish translator verbally tell you in English what a Spanish document says. Unless you change the document, you can use it without retranslation as much as you like.

Perhaps this is a good point to dispel a widespread myth: JavaScript is not the script version of the Java language. In fact, although they share the same name, that's virtually all they do share. Particularly good news is that JavaScript is much, much easier to learn and use than Java. In fact, languages like JavaScript are the easiest of all languages to learn, but they are still surprisingly powerful.

JavaScript and the Web

For most of this book you'll look at JavaScript code that runs inside a web page loaded into a browser. All you need in order to create these web pages is a text editor — for example, Windows Notepad — and a web browser, such as Firefox or Internet Explorer, with which you can view your pages. These browsers come equipped with JavaScript interpreters.

In fact, the JavaScript language first became available in the web browser Netscape Navigator 2. Initially, it was called LiveScript. However, because Java was the hot technology of the time, Netscape decided that JavaScript sounded more exciting. When JavaScript really took off, Microsoft decided to add its own brand of JavaScript, called JScript, to Internet Explorer. Since then, Netscape, Microsoft, and others have released improved versions and included them in their latest browsers. Although these different brands and versions of JavaScript have much in common, there are enough differences to cause problems if you're not careful. Initially you'll be creating code that'll work with most browsers, whether

Chapter 1: Introduction to JavaScript and the Web

Firefox, Internet Explorer, or Safari. Later chapters look at features available only to current browsers like Firefox 3 or later and Internet Explorer 7 and 8. You'll look into the problems with different browsers and versions of JavaScript later in this chapter and see how to deal with them.

You'll sometimes hear JavaScript referred to as ECMAScript. The ECMA (European Computer Manufacturers Association) is a private organization that develops standards in information and communication systems. One of the standards they control is for JavaScript, which they call ECMAScript. Their standard controls various aspects of the language and helps ensure that different versions of JavaScript are compatible. However, while the ECMA sets standards for the actual language, they don't specify how it's used in particular hosts. By *host*, we mean hosting environment; in this book, that will be the web browser. Other hosting environments include PDF files, web servers, Macromedia Flash applications, and many, many other places. In this book, we discuss only its use within the web browser. The organization that sets the standards for web pages is the World Wide Web Consortium (W3C). They not only set standards for HTML, XHTML, and XML, but also for how JavaScript interacts with web pages inside a web browser. You'll learn much more about this in later chapters of the book. Initially, you'll look at the essentials of JavaScript before the more advanced stuff. In the appendices of this book, you'll find useful guides to the JavaScript language and how it interacts with the web browser.

The majority of the web pages containing JavaScript that you create in this book can be stored on your hard drive and loaded directly into your browser from the hard drive itself, just as you'd load any normal file (such as a text file). However, this is not how web pages are loaded when you browse web sites on the Internet. The Internet is really just one great big network connecting computers. Access to web sites is a special service provided by particular computers on the Internet; the computers providing this service are known as *web servers*.

Basically, the job of a web server is to hold lots of web pages on its hard drive. When a browser, usually on a different computer, requests a web page contained on that web server, the web server loads it from its own hard drive and then passes the page back to the requesting computer via a special communications protocol called *Hypertext Transfer Protocol (HTTP)*. The computer running the web browser that makes the request is known as the *client*. Think of the client/server relationship as a bit like a customer/shopkeeper relationship. The customer goes into a shop and says, "Give me one of those." The shopkeeper serves the customer by reaching for the item requested and passing it back to the customer. In a web situation, the client machine running the web browser is like the customer, and the web server providing the page requested is like the shopkeeper.

When you type an address into the web browser, how does it know which web server to get the page from? Well, just as shops have addresses, say, 45 Central Avenue, Sometownsville, so do web servers. Web servers don't have street names; instead, they have *Internet protocol (IP) addresses*, which uniquely identify them on the Internet. These consist of four sets of numbers, separated by dots (for example, 127.0.0.1).

If you've ever surfed the net, you're probably wondering what on earth I'm talking about. Surely web servers have nice `www.somewebsite.com` names, not IP addresses? In fact, the `www.somewebsite.com` name is the "friendly" name for the actual IP address; it's a whole lot easier for us humans to remember. On the Internet, the friendly name is converted to the actual IP address by computers called *domain name servers*, which your Internet service provider will have set up for you.

One last thing: Throughout this book, we'll be referring to the Internet Explorer browser as IE.

Why Choose JavaScript?

JavaScript is not the only scripting language; there are others such as VBScript and Perl. So why choose JavaScript over the others?

The main reason for choosing JavaScript is its widespread use and availability. Both of the most commonly used browsers, IE and Firefox, support JavaScript, as do almost all of the less commonly used browsers. So you can assume that most people browsing your web site will have a version of JavaScript installed, though it is possible to use a browser's options to disable it.

Of the other scripting languages already mentioned, VBScript, which can be used for the same purposes as JavaScript, is supported only by Internet Explorer running on the Windows operating system, and Perl is not used at all in web browsers.

JavaScript is also very versatile and not just limited to use within a web page. For example, it can be used in Windows to automate computer-administration tasks and inside Adobe Acrobat PDF files to control the display of the page just as in web pages, although Acrobat uses a more limited version of JavaScript. However, the question of which scripting language is more powerful and useful has no real answer. Pretty much everything that can be done in JavaScript can be done in VBScript, and vice versa.

What Can JavaScript Do for Me?

The most common uses of JavaScript are interacting with users, getting information from them, and validating their actions. For example, say you want to put a drop-down menu on the page so that users can choose where they want to go to on your web site. The drop-down menu might be plain old HTML, but it needs JavaScript behind it to actually do something with the user's input. Other examples of using JavaScript for interactions are given by forms, which are used for getting information from the user. Again, these may be plain HTML, but you might want to check the validity of the information that the user is entering. For example, if you had a form taking a user's credit card details in preparation for the online purchase of goods, you'd want to make sure he had actually filled in those details before you sent the goods. You might also want to check that the data being entered are of the correct type, such as a number for his age rather than text.

JavaScript can also be used for various tricks. One example is switching an image in a page for a different one when the user rolls her mouse over it, something often seen in web page menus. Also, if you've ever seen scrolling messages in the browser's status bar (usually at the bottom of the browser window) or inside the page itself and wondered how that works, this is another JavaScript trick that you'll learn about later in the book. You'll also see how to create expanding menus that display a list of choices when a user rolls his or her mouse over them, another commonly seen JavaScript-driven trick.

Advances in browser sophistication and JavaScript mean that modern JavaScript is used for much more than a few clever tricks. In fact, quite advanced applications can be created. Examples of such applications include Google Maps, Google Calendar, and even a full-fledged word processor, Google Docs. These applications provide a real service. With a little inventiveness, you'll be amazed at what can be achieved. Of course, while JavaScript powers the user interface, the actual data processing is done in the background on powerful servers. JavaScript is powerful but still has limits.

Tools Needed to Create JavaScript Web Applications

The great news is that getting started learning JavaScript requires no expensive software purchases; you can learn JavaScript for free on any PC or Mac. You'll learn in this section what tools are available and how to obtain them.

Development Tools

All that you need to get started creating JavaScript code for web applications is a simple text editor, such as Windows Notepad, or one of the many slightly more advanced text editors that provide line numbering, search and replace, and so on. An alternative is a proper HTML editor; you'll need one that enables you to edit the HTML source code, because that's where you need to add your JavaScript. A number of very good tools specifically aimed at developing web-based applications, such as the excellent Dreamweaver from Adobe, are also available. However, this book concentrates on JavaScript rather than any specific development tool. When it comes to learning the basics, it's often best to write the code by hand rather than rely on a tool to do it for you. This helps you understand the fundamentals of the language before you attempt the more advanced logic that is beyond a tool's capability. When you have a good understanding of the basics, you can use tools as timesavers so that you can spend more time on the more advanced and more interesting coding.

Once you become more proficient, you may find that a web page editor makes life easier by inclusion of features such as checking the validity of your code, color-coding important JavaScript words, and making it easier to view your pages before loading them into a web browser. One example of free web development software is Microsoft's Visual Web Developer 2008 Express Edition, which you can download at <http://www.microsoft.com/express/vwd/>.

There are many other, equally good, free web page editors. A Google search on web editing software will bring back a long list of software you can use. Perhaps the most famous paid-for software is Adobe Dreamweaver.

As you write web applications of increasing complexity, you'll find useful tools that help you spot and solve errors. Errors in code are what programmers call bugs, though when our programs go wrong, we prefer to call them "unexpected additional features." Very useful in solving bugs are development tools called debuggers. Debuggers let you monitor what is happening in your code as it's running. In Chapter 4, you take an in-depth look at bugs and debugger development tools.

Web Browsers

In addition to software that lets you edit web pages, you'll also need a browser to view your web pages. It's best to develop your JavaScript code on the sort of browsers you expect visitors to use to access your web site. You'll see later in the chapter that although browsers are much more standards based, there are differences in how they view web pages and treat JavaScript code. All the examples provided in this book have been tested on Firefox version 3+ and IE versions 7 and 8. Wherever a piece of code does not work on any of these browsers, a note to this effect has been made in the text.

If you're running Windows, you'll almost certainly have IE installed. If not, a trip to <http://www.microsoft.com/windows/internet-explorer/default.aspx> will get you the latest version.

Firefox can be found at www.mozilla.com/firefox/all.html.

Chapter 1: Introduction to JavaScript and the Web

By default, most browsers have JavaScript support enabled. However, it is possible to disable this functionality in the browser. So before you start on your first JavaScript examples in the next section, you should check to make sure JavaScript is enabled in your browser.

To do this in Firefox, choose Tools ⇨ Options on the browser. In the window that appears, click the Content tab. From this tab, make sure the Enable JavaScript check box is selected, as shown in Figure 1-1.

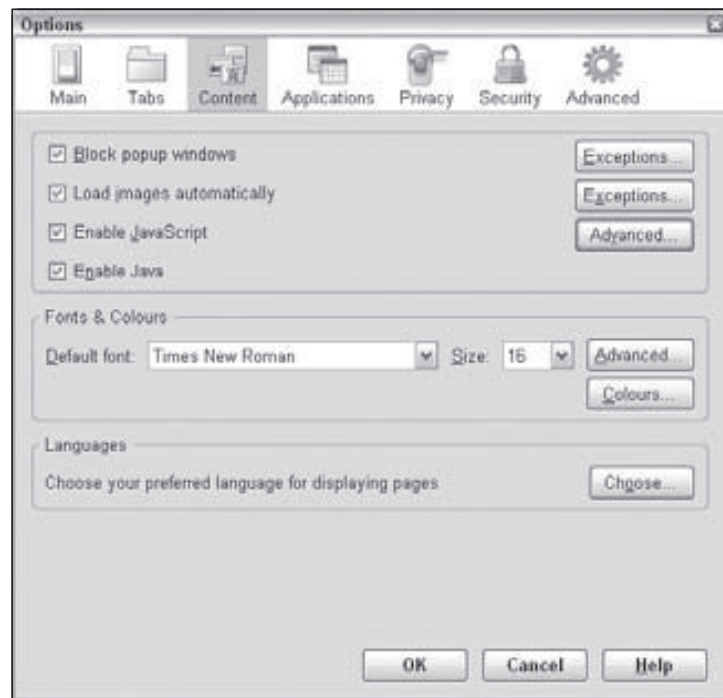


Figure 1-1

It is harder to turn off scripting in Internet Explorer. Choose Tools ⇨ Internet Options on the browser, click the Security tab, and check whether the Internet or Local intranet options have custom security settings. If either of them does, click the Custom Level button and scroll down to the Scripting section. Check that Active Scripting is set to Enable.

A final point to note is how to open the code examples in your browser. For this book, you simply need to open the file on your hard drive in which an example is stored. You can do this in a number of ways. One way in IE6 is to choose File ⇨ Open and click the Browse button to browse to where you stored the code. Similarly, in Firefox, choose File ⇨ Open File, browse to the file you want, and click the Choose File button.

IE7 and IE8, however, have a new menu structure, and this doesn't include an Open File option. You can get around this by typing the drive letter of your hard drive followed by a colon in the address bar (for example, C: for your C drive). In Microsoft Windows, you can press Ctrl+O for the Open file menu to appear. Alternatively, you can switch back to the Classic menu of earlier versions of IE. To do this, you can click Tools ⇨ Toolbars, and ensure the Menu Bar option is selected (see Figure 1-2).

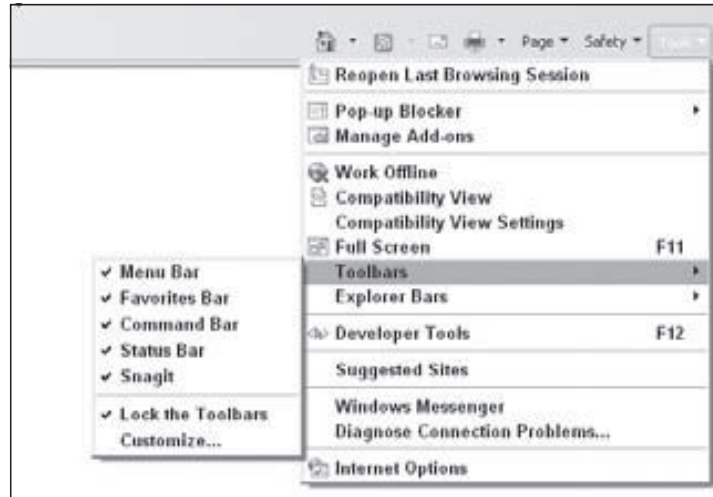


Figure 1-2

Where Do My Scripts Go?

Enough talk about the subject of JavaScript; it's time to look at how to put it into your web page. In this section, you'll find out how you add JavaScript code to your web page.

Including the `type` attribute is good practice, but within a web page it can be left off. Browsers such as IE and Firefox use JavaScript as their default script language. This means that if the browser encounters a `<script>` tag with no `type` attribute set, it assumes that the script block is written in JavaScript. However, use of the `type` attribute is specified as mandatory by W3C (the World Wide Web Consortium), which sets the standards for HTML.

Linking to an External JavaScript File

The `<script>` tag has another arrow in its quiver: the ability to specify that the JavaScript code is not inside the web page but inside a separate file. Any external files should be given the file extension `.js`. Though it's not compulsory, it does make it easier for you to work out what is contained in each of your files.

To link to an external JavaScript file, you need to create a `<script>` tag as described earlier and use its `src` attribute to specify the location of the external file. For example, imagine you've created a file called `MyCommonFunctions.js` that you wish to link to, and the file is in the same directory as your web page. The `<script>` tag would look like this:

```
<script type="text/javascript" src="MyCommonFunctions.js"></script>
```

The web browser will read this code and include the file contents as part of your web page. When linking to external files, you must not put any code within the `<script>` tags; for example, the following would be invalid:

```
<script type="text/javascript" src="MyCommonFunctions.js">
var myVariable;
```

Chapter 1: Introduction to JavaScript and the Web

```
if ( myVariable == 1 )
{
    // do something
}
</script>
```

If your web page is an XHTML document, you can omit the closing `</script>` tag and instead write this:

```
<script type="text/javascript" src="MyCommonFunctions.js" />
```

Generally, you use the `<script>` tag to load local files (those on the same computer as the web page itself). However, you can load external files from a web server by specifying the web address of the file. For example, if your file was called `MyCommonFunctions.js` and was loaded on a web server with the domain name `www.mysite.com`, the `<script>` tag would look like this:

```
<script type="text/javascript" src="http://www.mysite.com/MyCommonFunctions.js">
</script>
```

However, beware of linking to external files if they are controlled by other people. It would give those people the ability to control and change your web page, so you need to be very sure you trust them!

Advantages of Using an External File

The biggest advantage of external files is code reuse. Say you write a complex bit of JavaScript that performs a general function you might need in lots of pages. If you include the code inline (within the web page rather than via an external file), you need to cut and paste the code into each of your web pages that use it. This is fine as long as you never need to change the code, but the reality is you probably will need to change or improve the code at some point. If you've cut and pasted the code to 30 different web pages, you'll need to update it in 30 different places. Quite a headache! By using one external file and including it in all the pages that need it, you only need to update the code once and instantly all the 30 pages are updated. So much easier!

Another advantage of using external files is the browser will cache them, much as it does with images shared between pages. If your files are large, this could save download time and also reduce bandwidth usage.

Your First Simple JavaScript Programs

Enough talk about the subject of JavaScript; it's time to look at how to put it into your web page. In this section, you write your first piece of JavaScript code.

Inserting JavaScript into a web page is much like inserting any other HTML content; you use tags to mark the start and end of your script code. The tag used to do this is `<script>`. This tells the browser that the following chunk of text, bounded by the closing `</script>` tag, is not HTML to be displayed but rather script code to be processed. The chunk of code surrounded by the `<script>` and `</script>` tags is called a *script block*.

Basically, when the browser spots `<script>` tags, instead of trying to display the contained text to the user, it uses the browser's built-in JavaScript interpreter to run the code's instructions. Of course, the code might give instructions about changes to the way the page is displayed or what is shown in the page, but the text of the code itself is never shown to the user.

You can put the `<script>` tags inside the header (between the `<head>` and `</head>` tags) or inside the body (between the `<body>` and `</body>` tags) of the HTML page. However, although you can put them outside these areas — for example, before the `<html>` tag or after the `</html>` tag — this is not permitted in the web standards and so is considered bad practice.

The `<script>` tag has a number of attributes, but the most important one is `type`. As you saw earlier, JavaScript is not the only scripting language available, and different scripting languages need to be processed in different ways. You need to tell the browser which scripting language to expect so that it knows how to process that language. Your opening script tag will look like this:

```
<script type="text/javascript">
```

Including the `type` attribute is good practice, but within a web page it can be left off. Browsers such as IE and Firefox use JavaScript as their default script language. This means that if the browser encounters a `<script>` tag with no `type` attribute set, it assumes that the script block is written in JavaScript. However, use of the `type` attribute is specified as mandatory by W3C, which sets the standards for HTML.

Okay, let's take a look at the first page containing JavaScript code.

Try It Out Painting the Page Red

This is a simple example of using JavaScript to change the background color of the browser. In your text editor (we're using Windows Notepad), type the following:

```
<html>
<body bgcolor="WHITE">
<p>Paragraph 1</p>
<script type="text/javascript">
    document.bgColor = "RED";
</script>
</body>
</html>
```

Save the page as `ch1_examp1.htm` to a convenient place on your hard drive. Now load it into your web browser. You should see a red web page with the text `Paragraph 1` in the top-left corner. But wait — don't you set the `<body>` tag's `BGCOLOR` attribute to white? Okay, let's look at what's going on here.

The page is contained within `<html>` and `</html>` tags. This block contains a `<body>` element. When you define the opening `<body>` tag, you use HTML to set the page's background color to white.

```
<body bgcolor="WHITE">
```

Then you let the browser know that your next lines of code are JavaScript code by using the `<script>` start tag.

```
<script type="text/javascript">
```

Chapter 1: Introduction to JavaScript and the Web

Everything from here until the close tag, `</script>`, is JavaScript and is treated as such by the browser. Within this script block, you use JavaScript to set the document's background color to red.

```
document.backgroundColor = "RED";
```

What you might call the *page* is known as the *document* for the purpose of scripting in a web page. The document has lots of properties, including its background color, `backgroundColor`. You can reference properties of the document by writing `document`, followed by a dot, followed by the property name. Don't worry about the use of `document` at the moment; you look at it in greater depth later in the book.

Note that the preceding line of code is an example of a JavaScript *statement*. Every line of code between the `<script>` and `</script>` tags is called a statement, although some statements may run on to more than one line.

You'll also see that there's a semicolon (;) at the end of the line. You use a semicolon in JavaScript to indicate the end of a statement. In practice, JavaScript is very relaxed about the need for semicolons, and when you start a new line, JavaScript will usually be able to work out whether you mean to start a new line of code. However, for good coding practice, you should use a semicolon at the end of statements of code, and a single JavaScript statement should fit onto one line rather than continue on to two or more lines. Moreover, you'll find there are times when you must include a semicolon, which you'll come to later in the book.

Finally, to tell the browser to stop interpreting your text as JavaScript and start interpreting it as HTML, you use the script close tag:

```
</script>
```

You've now looked at how the code works, but you haven't looked at the order in which it works. When the browser loads in the web page, the browser goes through it, rendering it tag by tag from top to bottom of the page. This process is called *parsing*. The web browser starts at the top of the page and works its way down to the bottom of the page. The browser comes to the `<body>` tag first and sets the document's background to white. Then it continues parsing the page. When it comes to the JavaScript code, it is instructed to change the document's background to red.

Try It Out The Way Things Flow

Let's extend the previous example to demonstrate the parsing of a web page in action. Type the following into your text editor:

```
<html>
<body bgcolor="WHITE">
<p>Paragraph 1</p>
<script type="text/javascript">
  // Script block 1
  alert("First Script Block");
</script>
<p>Paragraph 2</p>
<script type="text/javascript">
  // Script block 2
  document.backgroundColor = "RED";
  alert("Second Script Block");
```

```
</script>
<p>Paragraph 3</p>
</body>
</html>
```

Save the file to your hard drive as `ch1_examp2.htm` and then load it into your browser. When you load the page, you should see the first paragraph, `Paragraph 1`, followed by a message box displayed by the first script block. The browser halts its parsing until you click the OK button. As you see in Figure 1-3, the page background is white, as set in the `<body>` tag, and only the first paragraph is displayed.

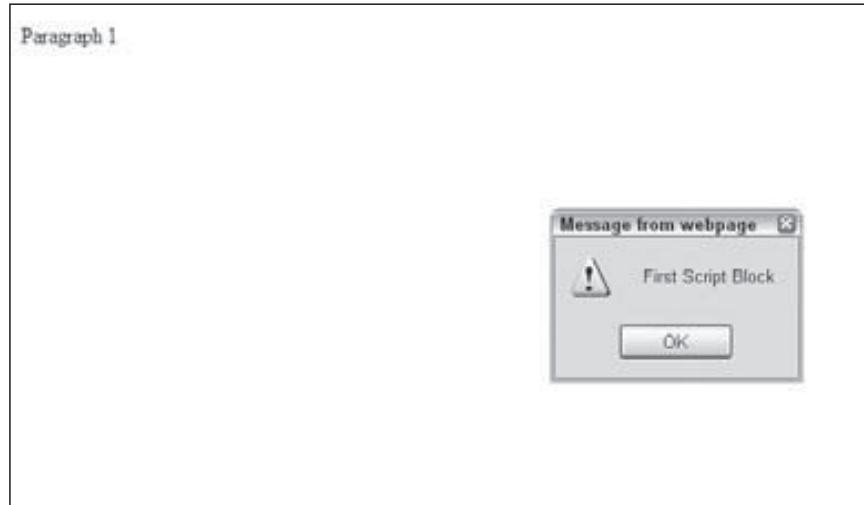


Figure 1-3

Click the OK button, and the parsing continues. The browser displays the second paragraph, and the second script block is reached, which changes the background color to red. Another message box is displayed by the second script block, as shown in Figure 1-4.

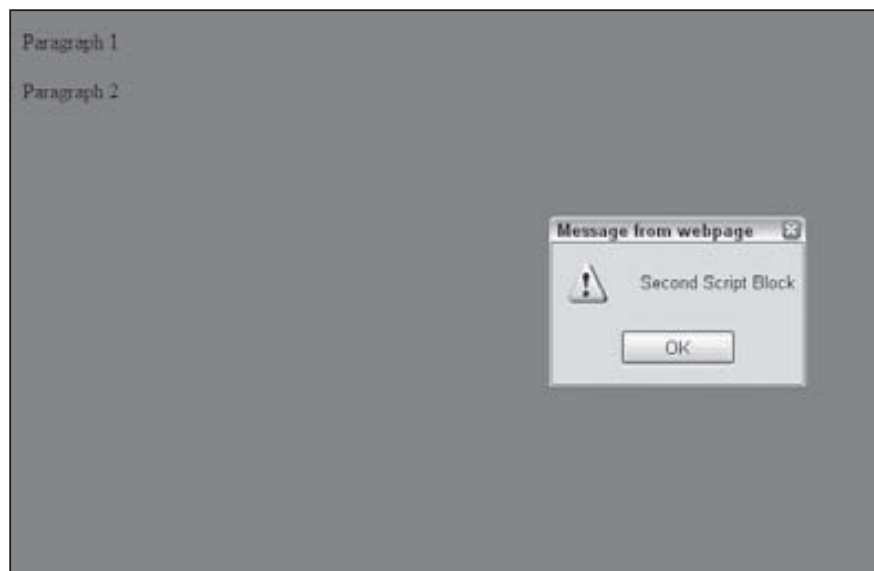


Figure 1-4

Chapter 1: Introduction to JavaScript and the Web

Click OK, and again the parsing continues, with the third paragraph, `Paragraph 3`, being displayed. The web page is complete, as shown in Figure 1-5.

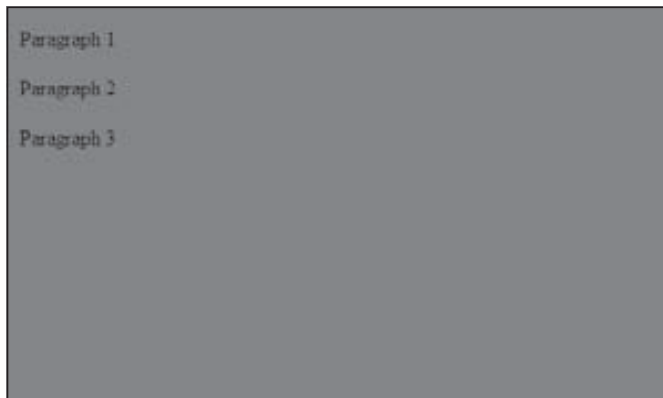


Figure 1-5

The first part of the page is the same as in our earlier example. The background color for the page is set to white in the definition of the `<body>` tag, and then a paragraph is written to the page.

```
<html>
<body bgcolor="WHITE">
<p>Paragraph 1</p>
```

The first new section is contained in the first script block.

```
<script type="text/javascript">
  // Script block 1
  alert("First Script Block");
</script>
```

This script block contains two lines, both of which are new to you. The first line —

```
// Script block 1
```

is just a *comment*, solely for your benefit. The browser recognizes anything on a line after a double forward slash (`//`) to be a comment and does not do anything with it. It is useful for you as a programmer because you can add explanations to your code that make it easier to remember what you were doing when you come back to your code later.

The `alert()` function in the second line of code is also new to you. Before learning what it does, you need to know what a *function* is.

Functions are defined more fully in Chapter 3, but for now you need only think of them as pieces of JavaScript code that you can use to do certain tasks. If you have a background in math, you may already have some idea of what a function is: A *function* takes some information, processes it, and gives you a result. A function makes life easier for you as a programmer because you don't have to think about how the function does the task — you can just concentrate on when you want the task done.

In particular, the `alert()` function enables you to alert or inform the user about something by displaying a message box. The message to be given in the message box is specified inside the parentheses of the `alert()` function and is known as the function's *parameter*.

The message box displayed by the `alert()` function is *modal*. This is an important concept, which you'll come across again. It simply means that the message box won't go away until the user closes it by clicking the OK button. In fact, parsing of the page stops at the line where the `alert()` function is used and doesn't restart until the user closes the message box. This is quite useful for this example, because it enables you to demonstrate the results of what has been parsed so far: The page color has been set to white, and the first paragraph has been displayed.

When you click OK, the browser carries on parsing down the page through the following lines:

```
<p>Paragraph 2</p>
<script type="text/javascript">
  // Script block 2
  document.bgColor = "RED";
  alert("Second Script Block");
</script>
```

The second paragraph is displayed, and the second block of JavaScript is run. The first line of the script block code is another comment, so the browser ignores this. You saw the second line of the script code in the previous example — it changes the background color of the page to red. The third line of code is the `alert()` function, which displays the second message box. Parsing is brought to a halt until you close the message box by clicking OK.

When you close the message box, the browser moves on to the next lines of code in the page, displaying the third paragraph and finally ending the web page.

```
<p>Paragraph 3</p>
</body>
</html>
```

Another important point raised by this example is the difference between setting properties of the page, such as background color, via HTML and doing the same thing using JavaScript. The method of setting properties using HTML is *static*: A value can be set only once and never changed again by means of HTML. Setting properties using JavaScript enables you to dynamically change their values. The term *dynamic* refers to something that can be changed and whose value or appearance is not set in stone.

This example is just that, an example. In practice, if you want the page's background to be red, you can set the `<body>` tag's `BGCOLOR` attribute to "RED" and not use JavaScript at all. Where you want to use JavaScript is where you want to add some sort of intelligence or logic to the page. For example, if the user's screen resolution is particularly low, you might want to change what's displayed on the page; with JavaScript, you can do this. Another reason for using JavaScript to change properties might be for special effects — for example, making a page fade in from white to its final color.

Chapter 1: Introduction to JavaScript and the Web

Try it Out Displaying Results in a Web Page

In this final example, you'll discover how to write information directly to a web page using JavaScript. This proves more useful when you're writing the results of a calculation or text you've created using JavaScript, as you'll see in the next chapter. For now, you'll just write "Hello World!" to a blank page using JavaScript:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<body >

<p id="ResultsP"></p>

<script type="text/javascript">
    // Script block 1
    document.getElementById('ResultsP').innerHTML = 'Hello World!';
</script>

</body>
</html>
```

Save the page as `ch1_examp3.htm` to a convenient place on your hard drive. Now load it into your web browser and you'll see Hello World! in the page. Although it would be easier to use HTML to do the same thing, this technique will prove useful in later chapters.

The first part of the page is the same as in our earlier examples, except the following line has been added:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

This lets the web browser know that you're using XHTML, the standard used throughout this book. It doesn't actually make any difference to the code; it would work just fine without the extra lines.

Consider this line:

```
<p id="ResultsP"></p>
```

You'll notice the `<p>` tag has been given an `id` using the `id` attribute. This `id` must be unique in the web page, because it is used by the JavaScript to identify the specific HTML element in the following line:

```
document.getElementById('ResultsP').innerHTML = 'Hello World!';
```

Don't worry if this seems complex at the moment; you'll learn more about how this works in later chapters, especially Chapters 6 and 12. Basically, the code is saying, "Get me the document element with `id` `ResultsP` and set the HTML inside that element to Hello World!"

It's important in your example that the code accessing the paragraph is after the paragraph. Otherwise, the code would be attempting to access a paragraph before it existed in the page and would throw an error.

A Brief Look at Browsers and Compatibility Problems

You've seen in the preceding example that by using JavaScript you can change a web page's document background color using the `bgColor` property of the `document`. The example worked whether you used a Netscape or Microsoft browser, because both types of browsers support a `document` with a `bgColor` property. You can say that the example is *cross-browser compatible*. However, it's not always the case that the property or language feature available in one browser will be available in another browser. This is even sometimes the case between versions of the same browser.

The version numbers for Internet Explorer and Firefox browsers are usually written as a decimal number; for example, Firefox has a version 1.5. This book uses the following terminology to refer to these versions: By version 1.x we mean all versions starting with the number 1; by version 1.0+ we mean all versions with a number greater than or equal to 1.

One of the main headaches involved in creating web-based JavaScript is the differences between different web browsers, the level of HTML they support, and the functionality their JavaScript interpreters can handle. You'll find that in one browser you can move an image using just a couple of lines of code but that in another it'll take a whole page of code or even prove impossible. One version of JavaScript will contain a method to change text to uppercase, and another won't. Each new release of IE or Firefox browsers sees new and exciting features added to its HTML and JavaScript support. The good news is that to a much greater extent than ever before, browser creators are complying with standards set by organizations such as the W3C. Also, with a little ingenuity, you can write JavaScript that will work with both IE and Firefox browsers.

Which browsers you want to support really comes down to the browsers you think the majority of your web site's visitors, that is, your *user base*, will be using. This book is aimed at both IE7 and later and Firefox 2 and later.

If you want your web site to be professional, you need to somehow deal with older browsers. You could make sure your code is backward compatible — that is, it only uses features available in older browsers. However, you may decide that it's simply not worth limiting yourself to the features of older browsers. In this case you need to make sure your pages degrade gracefully. In other words, make sure that although your pages won't work in older browsers, they will fail in a way that means the user is either never aware of the failure or is alerted to the fact that certain features on the web site are not compatible with his or her browser. The alternative to degrading gracefully is for your code to raise lots of error messages, cause strange results to be displayed on the page, and generally make you look like an idiot who doesn't know what you're doing!

So how do you make your web pages degrade gracefully? You can do this by using JavaScript to determine which browser the web page is running in after it has been partially or completely loaded. You can use this information to determine what scripts to run or even to redirect the user to another page written to make best use of her particular browser. In later chapters, you see how to find out what features the browser supports and take appropriate action so that your pages work acceptably on as many browsers as possible.

Summary

At this point, you should have a feel for what JavaScript is and what it can do. In particular, this brief introduction covered the following:

- ❑ You looked into the process the browser follows when interpreting your web page. It goes through the page element by element (parsing) and acts upon your HTML tags and JavaScript code as it comes to them.
- ❑ Unlike many programming languages, JavaScript requires just a text editor to start creating code. Something like Windows Notepad is fine for getting started, though more extensive tools will prove valuable once you get more experience.
- ❑ JavaScript code is embedded into the web page itself, along with the HTML. Its existence is marked out by the use of `<script>` tags. As with HTML, script executes from the top of the page and works down to the bottom, interpreting and executing the code statement by statement.

2

Data Types and Variables

One of the main uses of computers is to process and display information. By processing, we mean the information is modified, interpreted, or filtered in some way by the computer. For example, on an online banking web site, a customer may request details of all moneys paid out from his account in the last month. Here the computer would retrieve the information, filter out any information not related to payments made in the last month, and then display what's left in a web page. In some situations, information is processed without being displayed, and at other times, information is obtained directly without being processed. For example, in a banking environment, regular payments may be processed and transferred electronically without any human interaction or display.

In computing, information is referred to as *data*. Data come in all sorts of forms, such as numbers, text, dates, and times, to mention just a few. In this chapter, you look specifically at how JavaScript handles data such as numbers and text. An understanding of how data are handled is fundamental to any programming language.

The chapter starts by looking at the various types of data JavaScript can process. Then you look at how you can store these data in the computer's memory so you can use them again and again in the code. Finally, you see how to use JavaScript to manipulate and process the data.

Types of Data in JavaScript

Data can come in many different forms, or *types*. You'll recognize some of the data types that JavaScript handles from the world outside programming — for example, numbers and text. Other data types are a little more abstract and are used to make programming easier; one example is the object data type, which you won't see in detail until Chapter 4.

Some programming languages are strongly typed. In these languages, whenever you use a piece of data, you need to explicitly state what sort of data you are dealing with, and use of those data must follow strict rules applicable to its type. For example, you can't add a number and a word together.

Chapter 2: Data Types and Variables

JavaScript, on the other hand, is a weakly typed language and a lot more forgiving about how you use different types of data. When you deal with data, you often don't need to specify type; JavaScript will work that out for itself. Furthermore, when you are using different types of data at the same time, JavaScript will work out behind the scenes what it is you're trying to do.

Given how easygoing JavaScript is about data, why talk about data types at all? Why not just cut to the chase and start using data without worrying about their type?

First of all, while JavaScript is very good at working out what data it's dealing with, there are occasions when it'll get things wrong or at least not do what you want it to do. In these situations, you need to make it explicit to JavaScript what sort of data type you intended and how it should be used. To do that, you first need to know a little bit about data types.

A second reason is that data types enable you to use data effectively in your code. The things that can be done with data and the results you'll get depend on the type of data being used, even if you don't specify explicitly what type it is. For example, although trying to multiply two numbers together makes sense, doing the same thing with text doesn't. Also, the result of adding numbers is very different from the result of adding text. With numbers you get the sum, but with text you get one big piece of text consisting of the other pieces joined together.

Let's take a brief look at some of the more commonly used data types: numerical, text, and Boolean. You will see how to use them later in the chapter.

Numerical Data

Numerical data come in two forms:

- ❑ Whole numbers, such as 145, which are also known as *integers*. These numbers can be positive or negative and can span a very wide range in JavaScript: -2^{53} to 2^{53} .
- ❑ Fractional numbers, such as 1.234, which are also known as *floating-point* numbers. Like integers, they can be positive or negative, and they also have a massive range.

In simple terms, unless you're writing specialized scientific applications, you're not going to face problems with the size of numbers available in JavaScript. Also, although you can treat integers and floating-point numbers differently when it comes to storing them, JavaScript actually treats them both as floating-point numbers. It kindly hides the detail from you so you generally don't need to worry about it. One exception is when you want an integer but you have a floating-point number, in which case you'll round the number to make it an integer. You'll take a look at rounding numbers later in this chapter.

Text Data

Another term for one or more characters of text is a *string*. You tell JavaScript that text is to be treated as text and not as code simply by enclosing it inside quote marks ("). For example, "Hello World" and "A" are examples of strings that JavaScript will recognize. You can also use the single quote marks ('), so 'Hello World' and 'A' are also examples of strings that JavaScript will recognize. However, you must end the string with the same quote mark that you started it with. Therefore, "A' is not a valid JavaScript string, and neither is 'Hello World'.

What if you want a string with a single quote mark in the middle, say a string like `Peter O'Toole`? If you enclose it in double quotes, you'll be fine, so `"Peter O'Toole"` is recognized by JavaScript. However, `'Peter O'Toole'` will produce an error. This is because JavaScript thinks that your text string is `Peter O` (that is, it treats the middle single quote as marking the end of the string) and falls over wondering what the `Toole'` is.

Another way around this is to tell JavaScript that the middle `'` is part of the text and is not indicating the end of the string. You do this by using the backslash character (`\`), which has special meaning in JavaScript and is referred to as an *escape character*. The backslash tells the browser that the next character is not the end of the string, but part of the text. So `'Peter O\'Toole'` will work as planned.

What if you want to use a double quote inside a string enclosed in double quotes? Well, everything just said about the single quote still applies. So `'Hello "Paul"'` works, but `"Hello "Paul" "` won't. However, `"Hello \"Paul\" "` will also work.

JavaScript has a lot of other special characters, which can't be typed in but can be represented using the escape character in conjunction with other characters to create *escape sequences*. These work much the same as in HTML. For example, more than one space in a row is ignored in HTML, so a space is represented by the term ` `. Similarly, in JavaScript there are instances where you can't use a character directly but must use an escape sequence. The following table details some of the more useful escape sequences.

Escape Sequences	Character Represented
<code>\b</code>	Backspace
<code>\f</code>	Form feed
<code>\n</code>	New line
<code>\r</code>	Carriage return
<code>\t</code>	Tab
<code>\'</code>	Single quote
<code>\"</code>	Double quote
<code>\\</code>	Backslash
<code>\xNN</code>	NN is a hexadecimal number that identifies a character in the Latin-1 character set.

The least obvious of these is the last, which represents individual characters by their character number in the Latin-1 character set rather than by their normal appearance. Let's pick an example: Say you wanted to include the copyright symbol (©) in your string. What would your string need to look like? The answer is `"\xA9 Paul Wilton"`.

Similarly, you can refer to characters using their Unicode escape sequence. These are written `\uNNNN`, where `NNNN` refers to the Unicode number for that particular character. For example, to refer to the copyright symbol using this method, you use the string `\u00A9`.

Chapter 2: Data Types and Variables

Boolean Data

The use of yes or no, positive or negative, and true or false is commonplace in the physical world. The idea of true and false is also fundamental to digital computers; they don't understand maybes, only true and false. In fact, the concept of "yes or no" is so useful it has its own data type in JavaScript: the *Boolean* data type. The Boolean type has two possible values: `true` for yes and `false` for no.

The purpose of Boolean data in JavaScript is just the same as in the world outside programming: They enable you to answer questions and make decisions based on the answer. For example, if you are asked, "Is this book about JavaScript?" you would hopefully answer, "Yes it is," or you might also say, "That's true." Similarly you might say, "If it's false that the subject of the book is JavaScript, then put it down." Here you have a Boolean logic statement (named after its inventor George Boole), which asks a question and then does something based on whether the answer is true or false. In JavaScript, you can use the same sort of Boolean logic to give our programs decision-making abilities. You'll be taking a more detailed look at Boolean logic in the next chapter.

Variables — Storing Data in Memory

Data can be stored either permanently or temporarily.

You will want to keep important data, such as the details of a person's bank account, in a permanent store. For example, when Ms. Bloggs takes ten dollars or pounds or euros out of her account, you want to deduct the money from her account and keep a permanent record of the new balance. Information like this might be stored in something called a *database*.

However, there are other cases where you don't want to permanently store data, but simply want to keep a temporary note of it. Let's look at an example. Say Ms. Bloggs has a loan from BigBadBank Inc., and she wants to find out how much is still outstanding on this loan. She goes to the online banking page for loans and clicks a link to find out how much she owes. This is data that will be stored permanently somewhere. However, suppose you also provide a facility for increasing loan repayments to pay off the loan early. If Ms. Bloggs enters an increased repayment amount into the text box on the web page, you might want to show how much sooner the loan will be paid. This will involve a few possibly complex calculations, so to make it easier, you want to write code that calculates the result in several stages, storing the result at each stage as you go along, before providing a final result. After you've done the calculation and displayed the results, there's no need to permanently store the results for each stage, so rather than use a database, you need to use something called a *variable*. Why is it called a variable? Well, perhaps because a variable can be used to store temporary data that can be altered, or varied.

Another bonus of variables is that unlike permanent storage, which might be saved to disk or magnetic tape, variables are held in the computer's memory. This means that it is much, much faster to store and retrieve the data.

So what makes variables good places for temporarily storing your data? Well, variables have a limited lifetime. When your visitors close the page or move to a new one, your variables are lost, unless you take some steps to save them somewhere.

Each variable is given a name so that you can refer to it elsewhere in your code. These names must follow certain rules.

As with much of JavaScript code, you'll find that variable names are case sensitive. For example, `myVariable` is not the same as `myvariable`. You'll find that this is a very easy way for errors to slip into your code, even when you become an expert at JavaScript.

Also, you can't use certain names and characters for your variable names. Names you can't use are called *reserved* words. Reserved words are words that JavaScript keeps for its own use (for example, the word `var` or the word `with`). Certain characters are also forbidden in variable names: for example, the ampersand (&) and the percent sign (%). You are allowed to use numbers in your variable names, but the names must not begin with numbers. So `101myVariable` is not okay, but `myVariable101` is. Let's look at some more examples.

Invalid names include:

- ☐ `with`
- ☐ `99variables`
- ☐ `my%Variable`
- ☐ `theGood&theBad`

Valid names include

- ☐ `myVariable99`
- ☐ `myPercent_Variable`
- ☐ `the_Good_and_the_Bad`

You may wish to use a naming convention for your variables (for example, one that describes what sort of data you plan to hold in the variable). You can notate your variables in lots of different ways — none are right or wrong, but it's best to stick with one of them. One common method is *Hungarian notation*, where the beginning of each variable name is a three-letter identifier indicating the data type. For example, you may start integer variable names with `int`, floating-point variable names with `flt`, string variable names with `str`, and so on. However, as long as the names you use make sense and are used consistently, it really doesn't matter what convention you choose.

Creating Variables and Giving Them Values

Before you can use a variable, you should declare its existence to the computer using the `var` keyword. This warns the computer that it needs to reserve some memory for your data to be stored in later. To declare a new variable called `myFirstVariable`, write the following:

```
var myFirstVariable;
```

Note that the semicolon at the end of the line is not part of the variable name but instead is used to indicate to JavaScript the end of a statement. This line is an example of a JavaScript statement.

Once declared, a variable can be used to store any type of data. As we mentioned earlier, many other programming languages, called strongly typed languages, require you to declare not only the variable but also the type of data, such as numbers or text, that will be stored. However, JavaScript is a weakly typed language; you don't need to limit yourself to what type of data a variable can hold.

Chapter 2: Data Types and Variables

You put data into your variables, a process called *assigning values* to your variables, by using the equals sign (=). For example, if you want your variable named `myFirstVariable` to hold the number 101, you would write this:

```
myFirstVariable = 101;
```

The equals sign has a special name when used to assign values to a variable; it's called the *assignment operator*.

Try It Out Declaring Variables

Let's look at an example in which a variable is declared, store some data in it, and finally access its contents. You'll also see that variables can hold any type of data, and that the type of data being held can be changed. For example, you can start by storing text and then change to storing numbers without JavaScript having any problems. Type the following code into your text editor and save it as `ch2_examp1.htm`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

<head>
</head>
<body>

<script type="text/javascript">

var myFirstVariable;

myFirstVariable = "Hello";
alert(myFirstVariable);

myFirstVariable = 54321;
alert(myFirstVariable);

</script>

</body>
</html>
```

As soon as you load this into your web browser, it should show an alert box with "Hello" in it, as shown in Figure 2-1. This is the content of the variable `myFirstVariable` at that point in the code.



Figure 2-1

Click OK and another alert box appears with 54321 in it, as shown in Figure 2-2. This is the new value you assigned to the variable `myFirstVariable`.



Figure 2-2

Within the script block, you first declare your variable.

```
var myFirstVariable;
```

Currently, its value is the `undefined` value because you've declared only its existence to the computer, not any actual data. It may sound odd, but `undefined` is an actual primitive value in JavaScript, and it enables you to do comparisons. (For example, you can check to see if a variable contains an actual value or if it has not yet been given a value, that is, if it is `undefined`.) However, in the next line you assign `myFirstVariable` a string value, namely the value `Hello`.

```
myFirstVariable = "Hello";
```

Here you have assigned the variable a *literal* value (that is, a piece of actual data rather than data obtained by a calculation or from another variable). Almost anywhere that you can use a literal string or number, you can replace it with a variable containing number or string data. You see an example of this in the next line of code, where you use your variable `myFirstVariable` in the `alert()` function that you saw in the last chapter.

```
alert(myFirstVariable);
```

This causes the first alert box to appear. Next you store a new value in your variable, this time a number.

```
myFirstVariable = 54321;
```

The previous value of `myFirstVariable` is lost forever. The memory space used to store the value is freed up automatically by JavaScript in a process called *garbage collection*. Whenever JavaScript detects that the contents of a variable are no longer usable, such as when you allocate a new value, it performs the garbage collection process and makes the memory available. Without this automatic garbage collection process, more and more of the computer's memory would be consumed, until eventually the computer would run out and the system would grind to a halt. However, garbage collection is not always as efficient as it should be and may not occur until another page is loaded.

Just to prove that the new value has been stored, use the `alert()` function again to display the variable's new contents.

```
alert(myFirstVariable);
```

Assigning Variables with the Value of Other Variables

You've seen that you can assign a variable with a number or string, but can you assign a variable with the data stored inside another variable? The answer is yes, very easily, and in exactly the same way as giving a variable a literal value. For example, if you have declared the two variables `myVariable` and `myOtherVariable` and have given the variable `myOtherVariable` the value 22, like this:

```
var myVariable;  
var myOtherVariable;  
myOtherVariable = 22;
```

then you can use the following line to assign `myVariable` the same value as `myOtherVariable` (that is, 22).

```
myVariable = myOtherVariable;
```

Try It Out Assigning Variables the Values of Other Variables

Let's look at another example, this time assigning variables the values of other variables.

1. Type the following code into your text editor and save it as `ch2_examp2.htm`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<html xmlns="http://www.w3.org/1999/xhtml">  
<body>  
  
  <script language="JavaScript" type="text/javascript">  
  
    var string1 = "Hello";  
    var string2 = "Goodbye";  
  
    alert(string1);  
    alert(string2);  
  
    string2 = string1;  
  
    alert(string1);  
    alert(string2);  
  
    string1 = "Now for something different";  
  
    alert(string1);  
    alert(string2);  
  
  </script>  
  
</body>  
</html>
```

2. Load the page into your browser, and you'll see a series of six alert boxes appear.
3. Click OK on each alert box to see the next alert. The first two show the values of `string1` and `string2` — Hello and Goodbye, respectively. Then you assign `string2` the value that's in `string1`. The next two alert boxes show the contents of `string1` and `string2`; this time both are Hello.
4. Finally, you change the value of `string1`. Note that the value of `string2` remains unaffected. The final two alert boxes show the new value of `string1` (Now for something different) and the unchanged value of `string2` (Hello).

The first thing you do in the script block is declare your two variables: `string1` and `string2`. However, notice that you have assigned them values at the same time that you have declared them. This is a shortcut, called *initializing*, that saves you typing too much code.

```
var string1 ="Hello";  
var string2 = "Goodbye";
```

Note that you can use this shortcut with all data types, not just strings. The next two lines show the current value of each variable to the user using the `alert()` function.

```
alert(string1);  
alert(string2);
```

Then you assign `string2` the value that's contained in `string1`. To prove that the assignment has really worked, you again show the user the contents of each variable using the `alert()` function.

```
string2 = string1;  
  
alert(string1);  
alert(string2);
```

Next, you set `string1` to a new value.

```
string1 = "Now for something different";
```

This leaves `string2` with its current value, demonstrating that `string2` has its own copy of the data assigned to it from `string1` in the previous step. You'll see in later chapters that this is not always the case. However, as a general rule, basic data types, such as text and numbers, are always copied when assigned, whereas more complex data types, like the objects you come across in Chapter 4, are actually shared and not copied. For example, if you have a variable with the string `Hello` and assign five other variables the value of this variable, you now have the original data and five independent copies of the data. However, if it was an object rather than a string and you did the same thing, you'd find you still have only one copy of the data, but that six variables share it. Changing the data using any of the six variable names would change them for all the variables.

Finally, the `alert()` function is used to show the current values of each variable.

```
alert(string1);  
alert(string2);
```

Using Data — Calculations and Basic String Manipulation

Now that you've seen how to cope with errors, you can get back to the main subject of this chapter: data and how to use them. You've seen how to declare variables and how they can store information, but so far you haven't done anything really useful with this knowledge — so just why would you want to use variables at all?

What variables enable you to do is temporarily hold information that you can use for processing in mathematical calculations, in building up text messages, or in processing words that the user has entered. Variables are a little bit like the Memory Store button on the average pocket calculator. Say you were adding up your finances. You might first add up all the money you needed to spend, and then store it in temporary memory. After you had added up all your money coming in, you could deduct the amount stored in the memory to figure out how much would be left over. Variables can be used in a similar way: You can first gain the necessary user input and store it in variables, and then you can do your calculations using the values obtained.

In this section you'll see how you can put the values stored in variables to good use in both number-crunching and text-based operations.

Numerical Calculations

JavaScript has a range of basic mathematical capabilities, such as addition, subtraction, multiplication, and division. Each of the basic math functions is represented by a symbol: plus (+), minus (-), star (*), and forward slash (/), respectively. These symbols are called *operators* because they operate on the values you give them. In other words, they perform some calculation or operation and return a result to us. You can use the results of these calculations almost anywhere you'd use a number or a variable.

Imagine you were calculating the total value of items on a shopping list. You could write this calculation as follows:

Total cost of shopping = 10 + 5 + 5

Or, if you actually calculate the sum, it's

Total cost of shopping = 20

Now let's see how to do this in JavaScript. In actual fact, it is very similar except that you need to use a variable to store the final total.

```
var TotalCostOfShopping;  
TotalCostOfShopping = 10 + 5 + 5;  
alert(TotalCostOfShopping);
```

First, you declare a variable, `TotalCostOfShopping`, to hold the total cost.

In the second line, you have the code `10 + 5 + 5`. This piece of code is known as an *expression*. When you assign the variable `TotalCostOfShopping` the value of this expression, JavaScript automatically

calculates the value of the expression (20) and stores it in the variable. Notice that the equals sign tells JavaScript to store the results of the calculation in the `TotalCostOfShopping` variable. This is called *assigning* the value of the calculation to the variable, which is why the single equals sign (=) is called the *assignment operator*.

Finally, you display the value of the variable in an `alert` box.

The operators for subtraction and multiplication work in exactly the same way. Division is a little different.

Try It Out Calculations

Let's take a look at an example using the division operator to see how it works.

1. Enter the following code and save it as `ch2_examp3.htm`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<body>

<script language="JavaScript" type="text/javascript">
var firstNumber = 15;
var secondNumber = 10;
var answer;
answer = 15 / 10;
alert(answer);

alert(15 / 10);

answer = firstNumber / secondNumber;
alert(answer);

</script>

</body>
</html>
```

2. Load this into your web browser. You should see a succession of three `alert` boxes, each containing the value 1.5. These values are the results of three calculations.
3. The first thing you do in the script block is declare your three variables and assign the first two of these variables values that you'll be using later.

```
var firstNumber = 15;
var secondNumber = 10;
var answer;
```

4. Next, you set the `answer` variable to the results of the calculation of the expression `15/10`. You show the value of this variable in an `alert` box.

```
answer = 15 / 10;
alert(answer);
```

This example demonstrates one way of doing the calculation, but in reality you'd almost never do it this way.

Chapter 2: Data Types and Variables

To demonstrate that you can use expressions in places you'd use numbers or variables, you show the results of the calculation of `15/10` directly by including it in the `alert()` function.

```
alert(15 / 10);
```

Finally, you do the same calculation, but this time using the two variables `firstNumber`, which was set to 15, and `secondNumber`, which was set to 10. You have the expression `firstNumber / secondNumber`, the result of which you store in our `answer` variable. Then, to prove it has all worked, you show the value contained in `answer` by using your friend the `alert()` function.

```
answer = firstNumber / secondNumber;  
alert(answer);
```

Most calculations will be done in the third way (that is, using variables, or numbers and variables, and storing the result in another variable). The reason for this is that if the calculation used literal values (actual values, such as `15 / 10`), then you might as well program in the result of the calculation, rather than force JavaScript to calculate it for you. For example, rather than writing `15 / 10`, you might as well just write `1.5`. After all, the more calculations you force JavaScript to do, the slower it will be, though admittedly just one calculation won't tax it too much.

Another reason for using the result rather than the calculation is that it makes code more readable. Which would you prefer to read in code: `1.5 * 45 - 56 / 67 + 2.567` or `69.231`? Still better, a variable named for example `PricePerKG`, makes code even easier to understand for someone not familiar with it.

Increment and Decrement Operators

A number of operations using the math operators are so commonly used that they have been given their own operators. The two you'll be looking at here are the *increment* and *decrement* operators, which are represented by two plus signs (`++`) and two minus signs (`--`), respectively. Basically, all they do is increase or decrease a variable's value by one. You could use the normal `+` and `-` operators to do this, for example:

```
myVariable = myVariable + 1;  
myVariable = myVariable - 1;
```

You can assign a variable a new value that is the result of an expression involving its previous value.

However, using the increment and decrement operators shortens this to

```
myVariable++;  
myVariable--;
```

The result is the same — the value of `myVariable` is increased or decreased by one — but the code is shorter. When you are familiar with the syntax, this becomes very clear and easy to read.

Right now, you may well be thinking that these operators sound as useful as a poke in the eye. However, in Chapter 3, when you look at how you can run the same code a number of times, you'll see that these operators are very useful and widely used. In fact, the `++` operator is so widely used it has a computer language named after it: C++. The joke here is that C++ is one up from C. (Well, that's programmer humor for you!)

As well as placing the `++` or `--` after the variable, you can also place it before, like so:

```
++myVariable;  
--myVariable;
```

When the `++` and `--` are used on their own, as they usually are, it makes no difference where they are placed, but it is possible to use the `++` and `--` operators in an expression along with other operators. For example:

```
myVar = myNumber++ - 20;
```

This code takes 20 away from `myNumber` and then increments the variable `myNumber` by one before assigning the result to the variable `myVar`. If instead you place the `++` before and prefix it like this:

```
myVar = ++myNumber - 20;
```

First, `myNumber` is incremented by one, and then `myNumber` has 20 subtracted from it. It's a subtle difference but in some situations a very important one. Take the following code:

```
myNumber = 1;  
myVar = (myNumber++ * 10 + 1);
```

What value will `myVar` contain? Well, because the `++` is postfix (it's after the `myNumber` variable), it will be incremented afterwards. So the equation reads: Multiply `myNumber` by 10 plus 1 and then increment `myNumber` by one.

```
myVar = 1 * 10 + 1 = 11
```

Then add 1 to `myNumber` to get 12, but this is done after the value 11 has been assigned to `myVar`. Now take a look at the following code:

```
myNumber = 1;  
myVar = (++myNumber * 10 + 1);
```

This time `myNumber` is incremented by one first, then times 10 and plus 1.

```
myVar = 2 * 10 + 1 = 21
```

As you can imagine, such subtlety can easily be overlooked and lead to bugs in code; therefore, it's usually best to avoid this syntax.

Before going on, this seems to be a good point to introduce another operator: `+=`. This operator can be used as a shortcut for increasing the value held by a variable by a set amount. For example,

```
myVar += 6;
```

does exactly the same thing as

```
myVar = myVar + 6;
```

Chapter 2: Data Types and Variables

You can also do the same thing for subtraction and multiplication, as shown here:

```
myVar -= 6;  
myVar *= 6;
```

which is equivalent to

```
myVar = myVar - 6;  
myVar = myVar * 6;
```

Operator Precedence

You've seen that symbols that perform some function — like `+`, which adds two numbers together, and `-`, which subtracts one number from another — are called operators. Unlike people, not all operators are created equal; some have a higher *precedence* — that is, they get dealt with sooner. A quick look at a simple example will help demonstrate this point.

```
var myVariable;  
  
myVariable = 1 + 1 * 2;  
  
alert(myVariable);
```

If you were to type this, what result would you expect the `alert` box to show as the value of `myVariable`? You might expect that since $1 + 1 = 2$ and $2 * 2 = 4$, the answer is 4. Actually, you'll find that the `alert` box shows 3 as the value stored in `myVariable` as a result of the calculation. So what gives? Doesn't JavaScript add up right?

Well, you probably already know the reason from your understanding of mathematics. The way JavaScript does the calculation is to first calculate $1 * 2 = 2$, and then use this result in the addition, so that JavaScript finishes off with $1 + 2 = 3$.

Why? Because `*` has a higher precedence than `+`. The `=` symbol, also an operator (called the assignment operator), has the lowest precedence — it always gets left until last.

The `+` and `-` operators have an equal precedence, so which one gets done first? Well, JavaScript works from left to right, so if operators with equal precedence exist in a calculation, they get calculated in the order in which they appear when going from left to right. The same applies to `*` and `/`, which are also of equal precedence.

Try It Out Fahrenheit to Centigrade

Take a look at a slightly more complex example — a Fahrenheit to centigrade converter. (Centigrade is another name for the Celsius temperature scale.) Type this code and save it as `ch2_examp4.htm`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<html xmlns="http://www.w3.org/1999/xhtml">  
<body>
```

```
<script type="text/javascript">
// Equation is °C = 5/9 (°F - 32).
var degFahren = prompt("Enter the degrees in Fahrenheit",50);
var degCent;

degCent = 5/9 * (degFahren - 32);

alert(degCent);

</script>

</body>
</html>
```

If you load the page into your browser, you should see a prompt box, like that shown in Figure 2-3, that asks you to enter the degrees in Fahrenheit to be converted. The value 50 is already filled in by default.

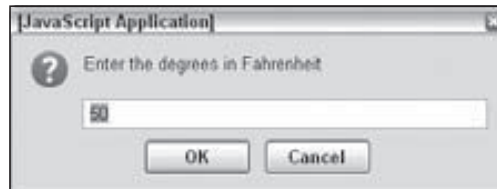


Figure 2-3

If you leave it at 50 and click OK, an alert box with the number 10 in it appears. This represents 50 degrees Fahrenheit converted to centigrade.

Reload the page and try changing the value in the prompt box to see what results you get. For example, change the value to 32 and reload the page. This time you should see 0 appear in the box.

As it's still a fairly simple example, there's no checking of data input so it'll let you enter abc as the degrees Fahrenheit. Later, in the "Data Type Conversion" section of this chapter, you'll see how to spot invalid characters posing as numeric data.

Try It Out Security Issues with Internet Explorer 8

When loading the page to Internet Explorer 8 (IE8), you may see the security warning issue shown in Figure 2-4, and the prompt window doesn't appear.



Figure 2-4

Chapter 2: Data Types and Variables

If it does you'll need change IE8's security settings and add file:///*.host as a trusted site. To do this:

1. Open IE8 and select the Internet Options menu from the Tools menu bar, as shown in Figure 2-5.

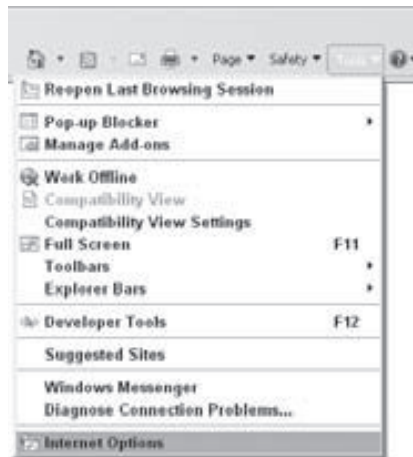


Figure 2-5

2. Click the Security tab and then click the green Trusted Sites button, as shown in Figure 2-6.



Figure 2-6

3. Click the Sites button and enter file:///*.host into the Add This Website to the Zone text box, as shown in Figure 2-7.

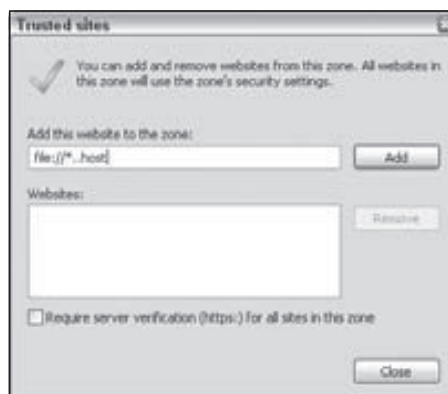


Figure 2-7

4. Make sure the Require Server Verification check box is unselected, click the Add button, and then click the Close button.
5. Click the OK button on the Internet Options dialog to return to the web page, refresh the page by pressing the F5 key, and the example will now work.

The first line of the script block is a comment, since it starts with two forward slashes (`//`). It contains the equation for converting Fahrenheit temperatures to centigrade and is in the example code solely for reference.

```
// Equation is °C = 5/9 (°F - 32).
```

Your task is to represent this equation in JavaScript code. You start by declaring your variables, `degFahren` and `degCent`.

```
var degFahren = prompt("Enter the degrees in Fahrenheit",50);  
var degCent;
```

Instead of initializing the `degFahren` variable to a literal value, you get a value from the user using the `prompt()` function. The `prompt()` function works in a similar way to an `alert()` function, except that as well as displaying a message, it also contains a text box in which the user can enter a value. It is this value that will be stored inside the `degFahren` variable. The value returned is a text string, but this will be implicitly converted by JavaScript to a number when you use it as a number, as discussed in the section on data type conversion later in this chapter.

You pass two pieces of information to the `prompt()` function:

- ❑ The text to be displayed — usually a question that prompts the user for input.
- ❑ The default value that is contained in the input box when the prompt dialog box first appears.

These two pieces of information must be specified in the given order and separated by a comma. If you don't want a default value to be contained in the input box when the prompt box opens, use an empty string (`" "`) for the second piece of information.

As you can see in the preceding code, the text is "Enter the degrees in Fahrenheit," and the default value in the input box is 50.

Next in the script block comes the equation represented in JavaScript. You store the result of the equation in the `degCent` variable. You can see that the JavaScript looks very much like the equation you have in the comment, except you use `degFahren` instead of `°F`, and `degCent` rather than `°C`.

```
degCent = 5/9 * (degFahren - 32);
```

The calculation of the expression on the right-hand side of the equals sign raises a number of important points. First, just as in math, the JavaScript equation is read from left to right, at least for the basic math functions like `+`, `-`, and so on. Secondly, as you saw earlier, just as there is precedence in math, there is in JavaScript.

Starting from the left, first JavaScript works out $5/9 = .5556$ (approximately). Then it comes to the multiplication, but wait . . . the last bit of our equation, `degFahren - 32`, is in parentheses. This raises the order of precedence and causes JavaScript to calculate the result of `degFahren - 32` before doing the multiplication. For example, when `degFahren` is set to 50, $(degFahren - 32) = (50 - 32) = 18$. Now JavaScript does the multiplication, $.5556 * 18$, which is approximately 10.

Chapter 2: Data Types and Variables

What if you didn't use the parentheses? Then your code would be

```
degCent = 5/9 * degFahren - 32;
```

The calculation of $5/9$ remains the same, but then JavaScript would have calculated the multiplication, $5/9 * \text{degFahren}$. This is because the multiplication takes precedence over the subtraction. When `degFahren` is 50, this equates to $5/9 * 50 = 27.7778$. Finally, JavaScript would have subtracted the 32, leaving the result as -4.2221 ; not the answer you want!

Finally, in your script block, you display the answer using the `alert()` function.

```
alert(degCent);
```

That concludes a brief look at basic calculations with JavaScript. However, in Chapter 4 you'll be looking at the `Math` object, which enables you to do more complex calculations.

Basic String Operations

In an earlier section, you looked at the text or string data type, as well as numerical data. Just as numerical data have associated operators, strings have operators too. This section introduces some basic string manipulation techniques using such operators. Strings are covered in more depth in Chapter 4, and advanced string handling is covered in Chapter 8.

One thing you'll find yourself doing again and again in JavaScript is joining two strings together to make one string — a process termed *concatenation*. For example, you may want to concatenate the two strings "Hello " and "Paul" to make the string "Hello Paul". So how do you concatenate? Easy! Use the `+` operator. Recall that when applied to numbers, the `+` operator adds them up, but when used in the context of two strings, it joins them together.

```
var concatString = "Hello " + "Paul";
```

The string now stored in the variable `concatString` is "Hello Paul". Notice that the last character of the string "Hello" is a space — if you left this out, your concatenated string would be "HelloPaul".

Try It Out Concatenating Strings

Let's look at an example using the `+` operator for string concatenation.

1. Type the following code and save it as `ch2_examp5.htm`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<body>

<script type="text/javascript">

var greetingString = "Hello";
var myName = prompt("Please enter your name", "");
var concatString;

document.write(greetingString + " " + myName + "<br>");
```

```
concatString = greetingString + " " + myName;

document.write(concatString);

</script>

</body>
</html>
```

2. If you load it into your web browser, you should see a prompt box asking for your name.
3. Enter your name and click OK. You should see a greeting and your name displayed twice on the web page.

You start the script block by declaring three variables. You set the first variable, `greetingString`, to a string value. The second variable, `myName`, is assigned to whatever is entered by the user in the prompt box. You do not initialize the third variable, `concatString`, here. It will be used to store the result of the concatenation that you'll do later in the code.

```
var greetingString = "Hello";
var myName = prompt("Please enter your name", "");
var concatString;
```

In the last chapter, you saw how the web page was represented by the concept of a document and that it had a number of different properties, such as `backgroundColor`. You can also use `document` to write text and HTML directly into the page itself. You do this by using the word `document`, followed by a dot, and then `write()`. You then use `document.write()` much as you do the `alert()` function, in that you put the text that you want displayed in the web page inside the parentheses following the word `write`. Don't worry too much about this here, though, because it will all be explained in detail in Chapter 4. However, you now make use of `document.write()` in your code to write the result of an expression to the page.

```
document.write(greetingString + " " + myName + "<br>");
```

The expression written to the page is the concatenation of the value of the `greetingString` variable, a space (" "), the value of the `myName` variable, and the HTML `
` tag, which causes a line break. For example, if you enter `Paul` into the prompt box, the value of this expression will be as follows:

```
Hello Paul<br>
```

In the next line of code is a similar expression. This time it is just the concatenation of the value in the variable `greetingString`, a space, and the value in the variable `myName`. You store the result of this expression in the variable `concatString`. Finally, you write the contents of the variable `concatString` to the page using `document.write()`.

```
concatString = greetingString + " " + myName;
document.write(concatString);
```

Mixing Numbers and Strings

What if you want to mix text and numbers in an expression? A prime example of this would be in the temperature converter you saw earlier. In the example, you just display the number without telling the user what it actually means. What you really want to do is display the number with descriptive text wrapped around it, such as "The value converted to degrees centigrade is 10."

Chapter 2: Data Types and Variables

Mixing numbers and text is actually very easy. You can simply join them together using the `+` operator. JavaScript is intelligent enough to know that when both a string and a number are involved, you're not trying to do numerical calculations, but rather that you want to treat the number as a string and join it to the text. For example, to join the text `My age is` and the number `101`, you could simply do the following:

```
alert("My age is " + 101);
```

This would produce an alert box with "My age is 101" inside it.

Try It Out Making the Temperature Converter User-Friendly

You can try out this technique of concatenating strings and numbers in our temperature-converter example. You'll output some explanatory text, along with the result of the conversion calculation. The changes that you need to make are very small, so load `ch2_examp4.htm` into your text editor and change the following line. Then save it as `ch2_examp6.htm`.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<body>

<script type="text/javascript">

var degFahren = prompt("Enter the degrees in Fahrenheit", 50);
var degCent;

degCent = 5/9 * (degFahren - 32);

alert(degFahren + "\xB0 Fahrenheit is " + degCent + "\xB0 centigrade");

</script>

</body>
</html>
```

Load the page into your web browser. Click OK in the prompt box to submit the value 50, and this time you should see the box shown in Figure 2-8.



Figure 2-8

This example is identical to `ch2_examp4.htm`, except for one line:

```
alert(degFahren + "\xB0 Fahrenheit is " + degCent + "\xB0 centigrade");
```

So we will just look at this line here. You can see that the `alert()` function contains an expression. Let's look at that expression more closely.

First is the variable `degFahren`, which contains numerical data. You concatenate that to the string `"\xB0 Fahrenheit is "`. JavaScript realizes that because you are adding a number and a string, you want to join them together into one string rather than trying to take their sum, and so automatically converts the number contained in `degFahren` to a string. You next concatenate this string to the variable `degCent`, containing numerical data. Again JavaScript converts the value of this variable to a string. Finally, you concatenate to the string `"\xB0 centigrade"`.

Note also the escape sequence used to insert the degree character into the strings. You'll remember from earlier in the chapter that `\xNN` can be used to insert special characters not available to type in directly. (*NN* is a hexadecimal number representing a character from the Latin-1 character table). So when JavaScript spots `\xB0` in a string, instead of showing those characters it does a lookup to see what character is represented by `B0` and shows that instead.

Something to be aware of when using special characters is that they are not necessarily cross-platform-compatible. Although you can use `\xNN` for a certain character on a Windows computer, you may find you need to use a different character on a Mac or a Unix machine.

You'll look at more string manipulation techniques in Chapter 4 — you'll see how to search strings and insert characters in the middle of them, and in Chapter 8 you'll see some very sophisticated string techniques.

Data Type Conversion

As you've seen, if you add a string and a number, JavaScript makes the sensible choice and converts the number to a string, then concatenates the two. Usually, JavaScript has enough sense to make data type conversions like this whenever it needs to, but there are some situations in which you need to convert the type of a piece of data yourself. For example, you may be given a piece of string data that you want to think of as a number. This is especially likely if you are using forms to collect data from the user. Any values input by the user are treated as strings, even though they may contain numerical data, such as the user's age.

Why is changing the type of the data so important? Consider a situation in which you collect two numbers from the user using a form and want to calculate their sum. The two numbers are available to you as strings, for example `"22"` and `"15"`. When you try to calculate the sum of these values using `"22" + "15"` you get the result `"2215"`, because JavaScript thinks you are trying to concatenate two strings rather than trying to find the sum of two numbers. To add to the possible confusion, the order also makes a difference. So:

```
1 + 2 + 'abc'
```

results in a string containing `"3abc"`, whereas:

```
'abc' + 1 + 2
```

would result in the string containing `"abc12"`.

Chapter 2: Data Types and Variables

In this section you'll look at two conversion functions that convert strings to numbers: `parseInt()` and `parseFloat()`.

Let's take `parseInt()` first. This function takes a string and converts it to an integer. The name is a little confusing at first — why `parseInt()` rather than `convertToInt()`? The main reason for the name comes from the way that the function works. It actually goes through (that is, parses) each character of the string you ask it to convert and sees if it's a valid number. If it is valid, `parseInt()` uses it to build up the number; if it is not valid, the command simply stops converting and returns the number it has converted so far.

For example, if your code is `parseInt("123")`, JavaScript will convert the string "123" to the number 123. For the code `parseInt("123abc")`, JavaScript will also return the number 123. When the JavaScript interpreter gets to the letter a, it assumes the number has ended and gives 123 as the integer version of the string "123abc".

The `parseFloat()` function works in the same way as `parseInt()`, except that it returns floating-point numbers — fractional numbers — and that a decimal point in the string, which it is converting, is considered to be part of the allowable number.

Try It Out Converting Strings to Numbers

Let's look at an example using `parseInt()` and `parseFloat()`. Enter the following code and save it as `ch2_examp7.htm`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<body>

<script type="text/javascript">

var myString = "56.02 degrees centigrade";
var myInt;
var myFloat;

document.write("\"" + myString + "\" is " + parseInt(myString) +
    " as an integer" + "<BR>");

myInt = parseInt(myString);
document.write("\"" + myString + "\" when converted to an integer equals " +
    myInt + "<BR>");

myFloat = parseFloat(myString);
document.write("\"" + myString +
    "\" when converted to a floating point number equals " + myFloat);

</script>

</body>
</html>
```

Load it into your browser, and you'll see three lines written in the web page, as shown in Figure 2-9.

"56.02 degrees centigrade" is 56 as an integer
"56.02 degrees centigrade" when converted to an integer equals 56
"56.02 degrees centigrade" when converted to a floating point number equals 56.02

Figure 2-9

Your first task in the script block is to declare some variables. The variable `myString` is declared and initialized to the string you want to convert. You could just as easily have used the string directly in this example rather than storing it in a variable, but in practice you'll find that you use variables more often than literal values. You also declare the variables `myInt` and `myFloat`, which will hold the converted numbers.

```
var myString = "56.02 degrees centigrade";  
var myInt;  
var myFloat;
```

Next, you write to the page the converted integer value of `myString` displayed inside a user-friendly sentence you build up using string concatenation. Notice that you use the escape sequence `\` to display quotes (") around the string you are converting.

```
document.write("\\" + myString + "\" is " + parseInt(myString) +  
  " as an integer" + "<BR>");
```

As you can see, you can use `parseInt()` and `parseFloat()` in the same places you would use a number itself or a variable containing a number. In fact, in this line the JavaScript interpreter is doing two conversions. First, it converts `myString` to an integer, because that's what you asked for by using `parseInt()`. Then it automatically converts that integer number back to a string, so it can be concatenated with the other strings to make up your sentence. Also note that only the 56 part of the `myString` variable's value is considered a valid number when you're dealing with integers. Anything after the 6 is considered invalid and is ignored.

Next, you do the same conversion of `myString` using `parseInt()`, but this time you store the result in the `myInt` variable. On the following line you use the result in some text you display to the user:

```
myInt = parseInt(myString);  
document.write("\\" + myString + "\" when converted to an integer equals " +  
  myInt + "<BR>");
```

Again, though `myInt` holds a number, the JavaScript interpreter knows that `+`, when a string and a number are involved, means you want the `myInt` value converted to a string and concatenated to the rest of the string so it can be displayed.

Finally, you use `parseFloat()` to convert the string in `myString` to a floating-point number, which you store in the variable `myFloat`. This time the decimal point is considered to be a valid part of the number, so it's anything after the 2 that is ignored. Again you use `document.write()` to write the result to the web page inside a user-friendly string.

```
myFloat = parseFloat(myString);  
document.write("\\" + myString + "\"  
  \" when converted to a floating point number equals " + myFloat);
```


Dealing with Strings That Won't Convert

Some strings simply are not convertible to numbers, such as strings that don't contain any numerical data. What happens if you try to convert these strings? As a little experiment, try changing the preceding example so that `myString` holds something that is not convertible. For example, change the line

```
var myString = "56.02 degrees centigrade";
```

to

```
var myString = "I'm a name not a number";
```

Now reload the page in your browser and you should see what's shown in Figure 2-10.

"I'm a name not a number" is NaN as an integer
"I'm a name not a number" when converted to an integer equals NaN
"I'm a name not a number" when converted to a floating point number equals NaN

Figure 2-10

You can see that in the place of the numbers you got before, you get `NaN`. What sort of number is that? Well, it's *Not a Number* at all!

If you use `parseInt()` or `parseFloat()` with any string that is empty or does not start with at least one valid digit, you get `NaN`, meaning Not a Number.

`NaN` is actually a special value in JavaScript. It has its own function, `isNaN()`, which checks whether something is `NaN` or not. For example,

```
myVar1 = isNaN("Hello");
```

will store the value `true` in the variable `myVar1`, since "Hello" is not a number, whereas

```
myVar2 = isNaN("34");
```

will store the value `false` in the variable `myVar2`, since 34 can be converted successfully from a string to a number by the `isNaN()` function.

In Chapter 3 you'll see how you can use the `isNaN()` function to check the validity of strings as numbers, something that proves invaluable when dealing with user input, as you'll see in Chapter 7.

Arrays

Now we're going to look at a new concept — something called an *array*. An array is similar to a normal variable, in that you can use it to hold any type of data. However, it has one important difference, which you'll see in this section.

As you have already seen, a normal variable can only hold one piece of data at a time. For example, you can set `myVariable` to be equal to 25 like so:

```
myVariable = 25;
```

and then go and set it to something else, say 35:

```
myVariable = 35;
```

However, when you set the variable to 35, the first value of 25 is lost. The variable `myVariable` now holds just the number 35.

The following table illustrates the variable:

Variable Name	Value
<code>myVariable</code>	35

The difference between such a normal variable and an array is that an array can hold *more than one* item of data at the same time. For example, you could use an array with the name `myArray` to store both the numbers 25 and 35. Each place where a piece of data can be stored in an array is called an *element*.

How do you distinguish between these two pieces of data in an array? You give each piece of data an *index* value. To refer to that piece of data you enclose its index value in square brackets after the name of the array. For example, an array called `myArray` containing the data 25 and 35 could be illustrated using the following table:

ElementName	Value
<code>myArray[0]</code>	25
<code>myArray[1]</code>	35

Notice that the index values start at 0 and not 1. Why is this? Surely 1 makes more sense — after all, we humans tend to say the first item of data, followed by the second item, and so on. Unfortunately, computers start from 0, and think of the first item as the zero item, the second as the first item, and so on. Confusing, but you'll soon get used to this.

Arrays can be very useful since you can store as many (within the limits of the language, which specifies a maximum of two to the power of 32 elements) or as few items of data in an array as you want. Also, you don't have to say up front how many pieces of data you want to store in an array, though you can if you wish.

So how do you create an array? This is slightly different from declaring a normal variable. To create a new array, you need to declare a variable name and tell JavaScript that you want it to be a new array using the `new` keyword and the `Array()` function. For example, the array `myArray` could be defined like this:

```
var myArray = new Array();
```

Chapter 2: Data Types and Variables

Note that, as with everything in JavaScript, the code is case-sensitive, so if you type `array()` rather than `Array()`, the code won't work. This way of defining an array will be explained further in Chapter 5.

As with normal variables, you can also declare your variable first, and then tell JavaScript you want it to be an array. For example:

```
var myArray;  
myArray = new Array();
```

Earlier you learned that you can say up front how many elements the array will hold if you want to, although this is not necessary. You do this by putting the number of elements you want to specify between the parentheses after `Array`. For example, to create an array that will hold six elements, you write the following:

```
var myArray = new Array(6);
```

You have seen how to declare a new array, but how do you store your pieces of data inside it? You can do this when you define your array by including your data inside the parentheses, with each piece of data separated by a comma. For example:

```
var myArray = new Array("Paul", 345, "John", 112, "Bob", 99);
```

Here the first item of data, "Paul", will be put in the array with an index of 0. The next piece of data, 345, will be put in the array with an index of 1, and so on. This means that the element with the name `myArray[0]` contains the value "Paul", the element with the name `myArray[1]` contains the value 345, and so on.

Note that you can't use this method to declare an array containing just one piece of numerical data, such as 345, because JavaScript assumes that you are declaring an array that will hold 345 elements.

This leads to another way of declaring data in an array. You could write the preceding line like this:

```
var myArray = new Array();  
myArray[0] = "Paul";  
myArray[1] = 345;  
myArray[2] = "John";  
myArray[3] = 112;  
myArray[4] = "Bob";  
myArray[5] = 99;
```

You use each element name as you would a variable, assigning them with values. You'll learn this method of declaring the values of array elements in the following "Try It Out" section.

Obviously, in this example the first way of defining the data items is much easier. However, there will be situations in which you want to change the data stored in a particular element in an array after they have been declared. In that case you will have to use the latter method of defining the values of the array elements.

You'll also spot from the preceding example that you can store different data types in the same array. JavaScript is very flexible as to what you can put in an array and where you can put it.

Before going on to an example, note here that if, for example, you had defined your array called `myArray` as holding three elements like this:

```
var myArray = new Array(3);
```

and then defined a value in the element with index 130 as follows:

```
myArray[130] = "Paul";
```

JavaScript would not complain and would happily assume that you had changed your mind and wanted an array that had (at least) 131 elements in it.

Try It Out An Array

In the following example, you'll create an array to hold some names. You'll use the second method described in the preceding section to store these pieces of data in the array. You'll then display the data to the user. Type the code and save it as `ch2_exam8.htm`.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<body>

<script type="text/javascript">

var myArray = new Array();
myArray[0] = "Bob";
myArray[1] = "Pete";
myArray[2] = "Paul";

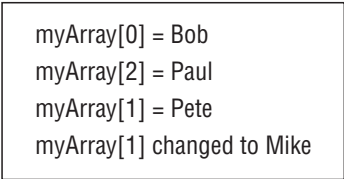
document.write("myArray[0] = " + myArray[0] + "<BR>");
document.write("myArray[2] = " + myArray[2] + "<BR>");
document.write("myArray[1] = " + myArray[1] + "<BR>");

myArray[1] = "Mike";
document.write("myArray[1] changed to " + myArray[1]);

</script>

</body>
</html>
```

If you load this into your web browser, you should see a web page that looks something like the one shown in Figure 2-11.



```
myArray[0] = Bob
myArray[2] = Paul
myArray[1] = Pete
myArray[1] changed to Mike
```

Figure 2-11

Chapter 2: Data Types and Variables

The first task in the script block is to declare a variable and tell the JavaScript interpreter you want it to be a new array.

```
var myArray = new Array();
```

Now that you have your array defined, you can store some data in it. Each time you store an item of data with a new index, JavaScript automatically creates a new storage space for it. Remember that the first element will be at `myArray[0]`.

Take each addition to the array in turn and see what's happening. Before you add anything, your array is empty. Then you add an array element with the following line:

```
myArray[0] = "Bob";
```

Your array now looks like this:

Index	Data Stored
0	Bob

Then you add another element to the array, this time with an index of 1.

```
myArray[1] = "Pete";
```

Index	Data Stored
0	Bob
1	Pete

Finally, you add another element to the array with an index of 2.

```
myArray[2] = "Paul";
```

Your array now looks like this:

Index	Data Stored
0	Bob
1	Pete
2	Paul

Next, you use a series of `document.write()` functions to insert the values that each element of the array contains into the web page. Here the array is out of order just to demonstrate that you can access it that way.

```
document.write("myArray[0] = " + myArray[0] + "<BR>");  
document.write("myArray[2] = " + myArray[2] + "<BR>");  
document.write("myArray[1] = " + myArray[1] + "<BR>");
```

You can treat each particular position in an array as if it's a standard variable. So you can use it to do calculations, transfer its value to another variable or array, and so on. However, if you try to access the data inside an array position before you have defined it, you'll get `undefined` as a value.

Finally, you change the value of the second array position to "Mike". You could have changed it to a number because, just as with normal variables, you can store any data type at any time in each individual data position in an array.

```
myArray[1] = "Mike";
```

Now your array's contents look like this:

Index	Data Stored
0	Bob
1	Mike
2	Paul

Just to show that the change you made has worked, you use `document.write()` to display the second element's value.

```
document.write("myArray[1] changed to " + myArray[1]);
```

A Multi-Dimensional Array

Suppose you want to store a company's personnel information in an array. You might have data such as names, ages, addresses, and so on. One way to create such an array would be to store the information sequentially — the first name in the first element of the array, then the corresponding age in the next element, the address in the third, the next name in the fourth element, and so on. Your array could look something like this:

Index	Data Stored
0	Name1
1	Age1
2	Address1
3	Name2
4	Age2
5	Address2
6	Name3
7	Age3
8	Address3

Chapter 2: Data Types and Variables

This would work, but there is a neater solution: using a *multi-dimensional array*. Up to now you have been using single-dimension arrays. In these arrays each element is specified by just one index — that is, one dimension. So, taking the preceding example, you can see Name1 is at index 0, Age1 is at index 1, and so on.

A multi-dimensional array is one with two or more indexes for each element. For example, this is how your personnel array could look as a two-dimensional array:

Index	0	1	2
0	Name1	Name2	Name3
1	Age1	Age2	Age3
2	Address1	Address2	Address3

You'll see how to create such multi-dimensional arrays in the following "Try It Out" section.

Try It Out A Two-Dimensional Array

The following example illustrates how you can create such a multi-dimensional array in JavaScript code and how you can access the elements of this array. Type the code and save it as `ch2_exam9.htm`.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<body>

<script type="text/javascript">

var personnel = new Array();

personnel[0] = new Array();
personnel[0][0] = "Name0";
personnel[0][1] = "Age0";
personnel[0][2] = "Address0";

personnel[1] = new Array();
personnel[1][0] = "Name1";
personnel[1][1] = "Age1";
personnel[1][2] = "Address1";

personnel[2] = new Array();
personnel[2][0] = "Name2";
personnel[2][1] = "Age2";
personnel[2][2] = "Address2";

document.write("Name : " + personnel[1][0] + "<BR>");
document.write("Age : " + personnel[1][1] + "<BR>");
document.write("Address : " + personnel[1][2]);

</script>
```



```
</body>  
</html>
```

If you load it into your web browser, you'll see three lines written into the page, which represent the name, age, and address of the person whose details are stored in the `personnel[1]` element of the array, as shown in Figure 2-12.

Name : Name1
Age : Age1
Address : Address1

Figure 2-12

The first thing to do in this script block is declare a variable, `personnel`, and tell JavaScript that you want it to be a new array.

```
var personnel = new Array();
```

Then you do something new; you tell JavaScript you want index 0 of the `personnel` array, that is, the element `personnel[0]`, to be another new array.

```
personnel[0] = new Array();
```

So what's going on? Well, the truth is that JavaScript doesn't actually support multi-dimensional arrays, only single ones. However, JavaScript enables us to fake multi-dimensional arrays by creating an array inside another array. So what the preceding line is doing is creating a new array inside the element with index 0 of our `personnel` array.

In the next three lines, you put values into the newly created `personnel[0]` array. JavaScript makes it easy to do this: You just state the name of the array, `personnel[0]`, followed by another index in square brackets. The first index (0) belongs to the `personnel` array; the second index belongs to the `personnel[0]` array.

```
personnel[0][0] = "Name0";  
personnel[0][1] = "Age0";  
personnel[0][2] = "Address0";
```

After these lines of code, your array looks like this:

Index	0
0	Name0
1	Age0
2	Address0

The numbers at the top, at the moment just 0, refer to the `personnel` array. The numbers going down the side, 0, 1, and 2, are actually indices for the new `personnel[0]` array inside the `personnel` array.

Chapter 2: Data Types and Variables

For the second person's details, you repeat the process, but this time you are using the `personnel` array element with index 1.

```
personnel[1] = new Array();
personnel[1][0] = "Name1";
personnel[1][1] = "Age1";
personnel[1][2] = "Address1";
```

Now your array looks like this:

Index	0	1
0	Name0	Name1
1	Age0	Age1
2	Address0	Address1

You create a third person's details in the next few lines. You are now using the element with index 2 inside the `personnel` array to create a new array.

```
personnel[2] = new Array();
personnel[2][0] = "Name2";
personnel[2][1] = "Age2";
personnel[2][2] = "Address2";
```

The array now looks like this:

Index	0	1	2
0	Name0	Name1	Name2
1	Age0	Age1	Age2
2	Address0	Address1	Address2

You have now finished creating your multi-dimensional array. You end the script block by accessing the data for the second person (`Name1`, `Age1`, `Address1`) and displaying it in the page by using `document.write()`. As you can see, accessing the data is very much the same as storing them. You can use the multi-dimensional array anywhere you would use a normal variable or single-dimension array.

```
document.write("Name : " + personnel[1][0] + "<BR>");
document.write("Age : " + personnel[1][1] + "<BR>");
document.write("Address : " + personnel[1][2]);
```

Try changing the `document.write()` commands so that they display the first person's details. The code would look like this:

```
document.write("Name : " + personnel[0][0] + "<BR>");
document.write("Age : " + personnel[0][1] + "<BR>");
document.write("Address : " + personnel[0][2]);
```

It's possible to create multi-dimensional arrays of three, four, or even a hundred dimensions, but things can start to get very confusing, and you'll find that you rarely, if ever, need more than two dimensions. To give you an idea, here's how to declare and access a five-dimensional array:

```
var myArray = new Array();
myArray[0] = new Array();
myArray[0][0] = new Array();
myArray[0][0][0] = new Array();
myArray[0][0][0][0] = new Array();

myArray[0][0][0][0][0] = "This is getting out of hand"

document.write(myArray[0][0][0][0][0]);
```

That's it for arrays for now, but you'll return to them in Chapter 4, where you'll find out something shocking about them. You'll also learn about some of their more advanced features.

Summary

In this chapter you have built up knowledge of the fundamentals of JavaScript's data types and variables and how to use them in operations. In particular, you saw that

- ❑ JavaScript supports a number of types of data, such as numbers, text, and Booleans.
- ❑ Text is represented by strings of characters and is surrounded by quotes. You must match the quotes surrounding strings. Escape characters enable you to include characters in your string that cannot be typed.
- ❑ Variables are JavaScript's means of storing data, such as numbers and text, in memory so that they can be used again and again in your code.
- ❑ Variable names must not include certain illegal characters, like the percent sign (%) and the ampersand (&), or be a reserved word, like `with`.
- ❑ Before you can give a value to a variable, you must declare its existence to the JavaScript interpreter.
- ❑ JavaScript has the four basic math operators, represented by the symbols plus (+), minus (-), star (*), and forward slash (/). To assign values of a calculation to a variable, you use the equals sign (=), termed the assignment operator.
- ❑ Operators have different levels of precedence, so multiplication and division will be calculated before addition and subtraction.
- ❑ Strings can be joined, or concatenated, to produce one big string by means of the + operator. When numbers and strings are concatenated with the + operator, JavaScript automatically converts the number into a string.
- ❑ Although JavaScript's automatic data conversion suits us most of the time, there are occasions when you need to force the conversion of data. You saw how `parseInt()` and `parseFloat()` can be used to convert strings to numbers. Attempting to convert strings that won't convert will result in `NaN` (Not a Number) being returned.
- ❑ Arrays are a special type of variable that can hold more than one piece of data. The data are inserted and accessed by means of a unique index number.

Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

1. Write a JavaScript program to convert degrees centigrade into degrees Fahrenheit, and to write the result to the page in a descriptive sentence. The JavaScript equation for Fahrenheit to centigrade is as follows:

```
degFahren = 9 / 5 * degCent + 32
```

2. The following code uses the `prompt()` function to get two numbers from the user. It then adds those two numbers together and writes the result to the page:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<script language="JavaScript" type="text/javascript">

var firstNumber = prompt("Enter the first number","");
var secondNumber = prompt("Enter the second number","");
var theTotal = firstNumber + secondNumber;
document.write(firstNumber + " added to " + secondNumber + " equals " +
    theTotal);

</script>
</body>
</html>
```

However, if you try the code out, you'll discover that it doesn't work. Why not? Change the code so that it does work.

3

Decisions, Loops, and Functions

So far, you've seen how to use JavaScript to get user input, perform calculations and tasks with that input, and write the results to a web page. However, a pocket calculator can do all this, so what is it that makes computers different? That is to say, what gives computers the appearance of having intelligence? The answer is the capability to make decisions based on information gathered.

How will decision-making help you in creating web sites? In the last chapter you wrote some code that converted temperature in degrees Fahrenheit to centigrade. You obtained the degrees Fahrenheit from the user using the `prompt()` function. This worked fine if the user entered a valid number, such as 50. If, however, the user entered something invalid for the Fahrenheit temperature, such as the string `aaa`, you would find that your code no longer works as expected. Now, if you had some decision-making capabilities in your program, you could check to see if what the user has entered is valid. If it is, you can do the calculation, and if it isn't, you can tell the user why and ask him to enter a valid number.

Validation of user input is probably one of the most common uses of decision making in JavaScript, but it's far from being the only use.

In this chapter you'll look at how decision making is implemented in JavaScript and how you can use it to make your code smarter.

Decision Making — The `if` and `switch` Statements

All programming languages enable you to make decisions — that is, they enable the program to follow a certain course of action depending on whether a particular *condition* is met. This is what gives programming languages their intelligence.

Chapter 3: Decisions, Loops, and Functions

For example, in a situation in which you use JavaScript code that is compatible only with version 4 or later browsers, the condition could be that the user is using a version 4 or later browser. If you discover that this condition is not met, you could direct him to a set of pages that are compatible with earlier browsers.

Conditions are comparisons between variables and data, such as the following:

- ❑ Is *A* bigger than *B*?
- ❑ Is *X* equal to *Y*?
- ❑ Is *M* not equal to *N*?

For example, if the variable `browserVersion` held the version of the browser that the user was using, the condition would be this:

Is `browserVersion` greater than or equal to 4?

You'll notice that all of these questions have a yes or no answer — that is, they are Boolean based and can only evaluate to `true` or `false`. How do you use this to create decision-making capabilities in your code? You get the browser to test for whether the condition is `true`. If (and only if) it is `true`, you execute a particular section of code.

Look at another example. Recall from Chapter 1 the natural English instructions used to demonstrate how code flows. One of these instructions for making a cup of coffee is:

Has the kettle boiled? If so, then pour water into cup; otherwise, continue to wait.

This is an example of making a decision. The condition in this instruction is “Has the kettle boiled?” It has a `true` or `false` answer. If the answer is `true`, you pour the water into the cup. If it isn't `true`, you continue to wait.

In JavaScript, you can change the flow of the code's execution depending on whether a condition is `true` or `false`, using an `if` statement or a `switch` statement. You will look at these shortly, but first we need to introduce some new operators that are essential for the definition of conditions — *comparison operators*.

Comparison Operators

In Chapter 2 you saw how mathematical functions, such as addition and division, were represented by symbols, such as plus (+) and forward slash (/), called operators. You also saw that if you want to give a variable a value, you can assign to it a value or the result of a calculation using the equals sign (=), termed the assignment operator.

Decision making also has its own operators, which enable you to test conditions. Comparison operators, just like the mathematical operators you saw in the last chapter, have a left-hand side (LHS) and a right-hand side (RHS), and the comparison is made between the two. The technical terms for these are the *left operand* and the *right operand*. For example, the less-than operator, with the symbol `<`, is a comparison operator. You could write `23 < 45`, which translates as “Is 23 less than 45?” Here, the answer would be `true` (see Figure 3-1).

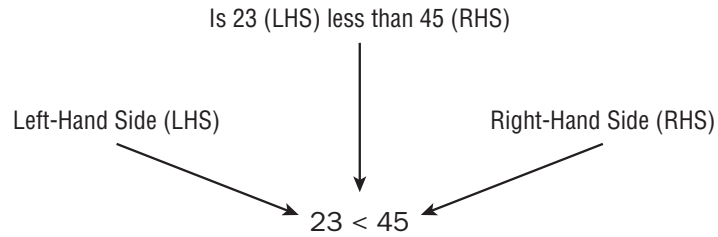


Figure 3-1

There are other comparison operators, the more useful of which are summarized in the following table:

Operator Symbol	Purpose
==	Tests if LHS is equal to RHS
<	Tests if LHS is less than RHS
>	Tests if LHS is greater than RHS
<=	Tests if LHS is less than or equal to RHS
>=	Tests if LHS is greater than or equal to RHS
!=	Tests if LHS is not equal to RHS

You'll see these comparison operators in use in the next section when you look at the `if` statement.

Precedence

Recall from Chapter 2 that operators have an order of precedence. This applies also to the comparison operators. The `==` and `!=` comparison operators have the lowest order of precedence, and the rest of the comparison operators, `<`, `>`, `<=`, and `>=`, have an equal precedence.

All of these comparison operators have a precedence that is below operators, such as `+`, `-`, `*`, and `/`. This means that if you make a comparison such as `3 * 5 > 2 * 5`, the multiplication calculations are worked out first, before their results are compared. However, in these circumstances, it's both safer and clearer if you wrap the calculations on either side inside parentheses, for example, `(3 * 5) > (2 * 5)`. As a general rule, it's a good idea to use parentheses to ensure that the precedence is clear, or you may find yourself surprised by the outcome.

Assignment versus Comparison

One very important point to mention is the ease with which the assignment operator (`=`) and the comparison operator (`==`) can be mixed up. Remember that the `=` operator assigns a value to a variable and that the `==` operator compares the value of two variables. Even when you have this idea clear, it's amazingly easy to put one equals sign where you meant to put two.

Chapter 3: Decisions, Loops, and Functions

Assigning the Results of Comparisons

You can store the results of a comparison in a variable, as shown in the following example:

```
var age = prompt("Enter age:", "");
var isOverSixty = parseInt(age) > 60;
document.write("Older than 60: " + isOverSixty);
```

Here you obtain the user's age using the `prompt()` function. This returns, as a string, whatever value the user enters. You then convert that to a number using the `parseInt()` function you saw in the previous chapter and use the greater-than operator to see if it's greater than 60. The result (either `true` or `false`) of the comparison will be stored in the variable `isOverSixty`.

If the user enters 35, the `document.write()` on the final line will write this to the page:

```
Older than 60: false
```

If the user entered 61, this will be displayed:

```
Older than 60: true
```

The if Statement

The `if` statement is one you'll find yourself using in almost every program that is more than a couple of lines long. It works very much as it does in the English language. For example, you might say in English, "If the room temperature is more than 80 degrees Fahrenheit, then I'll turn the air conditioning on." In JavaScript, this would translate into something like this:

```
if (roomTemperature > 80)
{
    roomTemperature = roomTemperature - 10;
}
```

How does this work? See Figure 3-2.

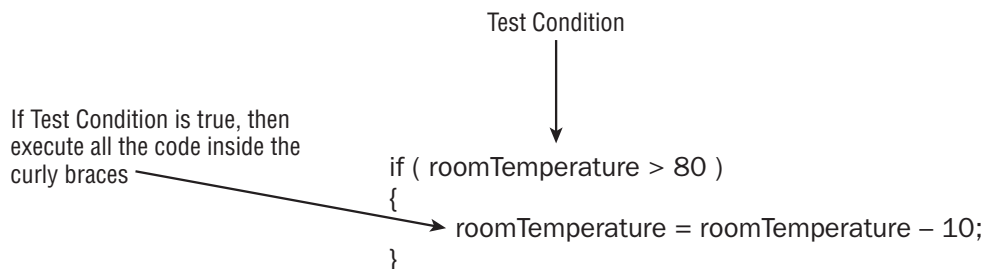


Figure 3-2

Notice that the test condition is placed in parentheses and follows the `if` keyword. Also, note that there is no semicolon at the end of this line. The code to be executed if the condition is `true` is placed in curly braces on the line after the condition, and each of these lines of code does end with a semicolon.

The curly braces, {}, have a special purpose in JavaScript: They mark out a *block* of code. Marking out lines of code as belonging to a single block means that JavaScript will treat them all as one piece of code. If the condition of an `if` statement is `true`, JavaScript executes the next line or block of code following the `if` statement. In the preceding example, the block of code has only one statement, so we could equally as well have written this:

```
if (roomTemperature > 80)
    roomTemperature = roomTemperature - 10;
```

However, if you have a number of lines of code that you want to execute, you need the braces to mark them out as a single block of code. For example, a modified version of the example with three statements of code would have to include the braces.

```
if (roomTemperature > 80)
{
    roomTemperature = roomTemperature - 10;
    alert("It's getting hot in here");
    alert("Air conditioning switched on");
}
```

A particularly easy mistake to make is to forget the braces when marking out a block of code to be executed. Instead of the code in the block being executed when the condition is true, you'll find that *only the first line* after the `if` statement is executed. However, the other lines will always be executed regardless of the outcome of the test condition. To avoid mistakes like these, it's a good idea to always use braces, even where there is only one statement. If you get into this habit, you'll be less likely to leave them out when they are actually needed.

Try It Out The if Statement

Let's return to the temperature converter example from Chapter 2 and add some decision-making functionality.

1. Enter the following code and save it as `ch3_examp1.htm`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<body>

<script type="text/javascript">

var degFahren = Number(prompt("Enter the degrees Fahrenheit",32));
var degCent;

degCent = 5/9 * (degFahren - 32);

document.write(degFahren + "\xB0 Fahrenheit is " + degCent +
    "\xB0 centigrade<br />");

if (degCent < 0)
{
    document.write("That's below the freezing point of water");
}
```

Chapter 3: Decisions, Loops, and Functions

```
}

if (degCent == 100)
    document.write("That's the boiling point of water");

</script>

</body>
</html>
```

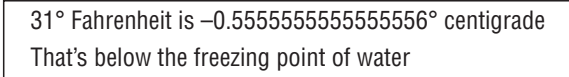
2. Load the page into your browser and enter 32 into the prompt box for the Fahrenheit value to be converted. With a value of 32, neither of the `if` statement's conditions will be true, so the only line written in the page will be that shown in Figure 3-3.



32° Fahrenheit is 0° centigrade

Figure 3-3

3. Now reload the page and enter 31 for the Fahrenheit value. This time you'll see two lines in the page, as shown in Figure 3-4.



31° Fahrenheit is -0.555555555555556° centigrade
That's below the freezing point of water

Figure 3-4

4. Finally, reload the page again, but this time, enter 212 in the prompt box. The two lines shown in Figure 3-5 will appear in the page.



212° Fahrenheit is 100° centigrade
That's the boiling point of water

Figure 3-5

The first part of the script block in this page is taken from the example `ch2_examp4.htm` in Chapter 2. You declare two variables, `degFahren` and `degCent`. The variable `degFahren` is given an initial value obtained from the user with the `prompt()` function. Note the `prompt()` function returns a string value, which you then explicitly convert to a numeric value using the `Number()` function. The variable `degCent` is then set to the result of the calculation `5/9 * (degFahren - 32)`, which is the Fahrenheit-to-centigrade conversion calculation.

```
var degFahren = Number(prompt("Enter the degrees Fahrenheit",32));
var degCent;

degCent = 5/9 * (degFahren - 32);
```

Then you write the result of your calculation to the page.

```
document.write(degFahren + "\xB0 Fahrenheit is " + degCent +
    "\xB0 centigrade<br />");
```

Now comes the new code; the first of two `if` statements.

```
if (degCent < 0)
{
    document.write("That's below the freezing point of water");
}
```

This `if` statement has the condition that asks, “Is the value of the variable `degCent` less than zero?” If the answer is yes (`true`), the code inside the curly braces executes. In this case, you write a sentence to the page using `document.write()`. If the answer is no (`false`), the processing moves on to the next line after the closing brace. Also worth noting is the fact that the code inside the `if` statement’s opening brace is indented. This is not necessary, but it is a good practice to get into because it makes your code much easier to read.

When trying out the example, you started by entering 32, so that `degFahren` will be initialized to 32. In this case the calculation `degCent = 5/9 * (degFahren - 32)` will set `degCent` to 0. So the answer to the question “Is `degCent` less than zero?” is `false`, because `degCent` is equal to zero, not less than zero. The code inside the curly braces will be skipped and never executed. In this case, the next line to be executed will be the second `if` statement’s condition, which we’ll discuss shortly.

When you entered 31 in the prompt box, `degFahren` was set to 31, so the variable `degCent` will be `-0.555555555556`. So how does your `if` statement look now? It evaluates to “Is `-0.555555555556` less than zero?” The answer this time is `true`, and the code inside the braces, here just a `document.write()` statement, executes.

Finally, when you entered 212, how did this alter the `if` statement? The variable `degCent` is set to 100 by the calculation, so the `if` statement now asks the question, “Is 100 less than zero?” The answer is `false`, and the code inside the braces will be skipped over.

In the second `if` statement, you evaluate the condition “Is the value of variable `degCent` equal to 100?”

```
if (degCent == 100)
    document.write("That's the boiling point of water");
```

There are no braces here, so if the condition is `true`, the only code to execute is the first line below the `if` statement. When you want to execute multiple lines in the case of the condition being `true`, braces are required.

You saw that when `degFahren` is 32, `degCent` will be 0. So your `if` statement will be “Is 0 equal to 100?” The answer is clearly `false`, and the code won’t execute. Again, when you set `degFahren` to 31, `degCent` will be calculated to be `-0.555555555556`; “Is `-0.555555555556` equal to 100?” is also `false`, and the code won’t execute.

Finally, when `degFahren` is set to 212, `degCent` will be 100. This time the `if` statement is “Is 100 equal to 100?” and the answer is `true`, so the `document.write()` statement executes.

As you have seen already, one of the most common errors in JavaScript, even for experts, is using one equals sign for evaluating, rather than the necessary two. Take a look at the following code extract:

```
if (degCent = 100)
    document.write("That's the boiling point of water");
```

This condition will always evaluate to `true`, and the code below the `if` statement will always execute. Worse still, your variable `degCent` will be set to 100. Why? Because a single equals sign assigns values to a variable; only a double equals sign compares values. The reason an assignment always evaluates to

Chapter 3: Decisions, Loops, and Functions

`true` is that the result of the assignment expression is the value of the right-hand side expression and this is the number `100`, which is then implicitly converted to a Boolean and any number besides `0` and `NaN` converts to `true`.

Logical Operators

You should have a general idea of how to use conditions in `if` statements now, but how do you use a condition such as “Is `degFahren` greater than zero but less than `100`?” There are two conditions to test here. You need to test whether `degFahren` is greater than zero *and* whether `degFahren` is less than `100`.

JavaScript enables you to use such multiple conditions. To do this, you need to learn about three more operators: the logical operators `AND`, `OR`, and `NOT`. The symbols for these are listed in the following table.

Operator	Symbol
AND	<code>&&</code>
OR	<code> </code>
NOT	<code>!</code>

Notice that the `AND` and `OR` operators are *two* symbols repeated: `&&` and `||`. If you type just one symbol, `&` or `|`, strange things will happen because these are special operators called *bitwise operators* used in binary operations — for logical operations you must always use two.

After you’ve learned about the three logical operators, you’ll take a look at how to use them in `if` statements, with plenty of practical examples. So if it seems a bit confusing on first read, don’t panic. All will become clear. Let’s look at how each of these works, starting with the `AND` operator.

AND

Recall that we talked about the left-hand side (LHS) and the right-hand side (RHS) of the operator. The same is true with the `AND` operator. However, now the LHS and RHS of the condition are Boolean values (usually the result of a condition).

The `AND` operator works very much as it does in English. For example, you might say, “If I feel cold *and* I have a coat, then I’ll put my coat on.” Here, the left-hand side of the “and” word is “Do I feel cold?” and this can be evaluated as `true` or `false`. The right-hand side is “Do I have a coat?” which again is evaluated to either `true` or `false`. If the left-hand side is `true` (I am cold) *and* the right-hand side is `true` (I do have a coat), then you put your coat on.

This is very similar to how the `AND` operator works in JavaScript. The `AND` operator actually produces a result, just as adding two numbers together produces a result. However, the `AND` operator takes two Boolean values (on its LHS and RHS) and results in another Boolean value. If the LHS and RHS conditions evaluate to `true`, the result will be `true`. In any other circumstance, the result will be `false`.

Following is a *truth table* of possible evaluations of left-hand sides and right-hand sides and the result when AND is used.

Left-Hand Side	Right-Hand Side	Result
true	true	true
false	true	false
true	false	false
false	false	false

Although the table is, strictly speaking, true, it's worth noting that JavaScript doesn't like doing unnecessary work. Well, who does! If the left-hand side is `false`, even if the right-hand side does evaluate to `true`, it won't make any difference to the final result — it'll still be `false`. So to avoid wasting time, if the left-hand side is `false`, JavaScript doesn't even bother checking the right-hand side and just returns a result of `false`.

OR

Just like AND, OR also works much as it does in English. For example, you might say that if it is raining *or* if it is snowing, then you'll take an umbrella. If either of the conditions "it is raining" or "it is snowing" is true, you will take an umbrella.

Again, just like AND, the OR operator acts on two Boolean values (one from its left-hand side and one from its right-hand side) and returns another Boolean value. If the left-hand side evaluates to `true` or the right-hand side evaluates to `true`, the result returned is `true`. Otherwise, the result is `false`. The following table shows the possible results.

Left-Hand Side	Right-Hand Side	Result
true	true	true
false	true	true
true	false	true
false	false	false

As with the AND operator, JavaScript likes to avoid doing things that make no difference to the final result. If the left-hand side is `true`, then whether the right-hand side is `true` or `false` makes no difference to the final result — it'll still be `true`. So, to avoid work, if the left-hand side is `true`, the right-hand side is not evaluated, and JavaScript simply returns `true`. The end result is the same — the only difference is in how JavaScript arrives at the conclusion. However, it does mean you should not rely on the right-hand side of the OR operator to be executed.

Chapter 3: Decisions, Loops, and Functions

NOT

In English, we might say, “If I’m *not* hot, then I’ll eat soup.” The condition being evaluated is whether we’re hot. The result is true or false, but in this example we act (eat soup) if the result is false.

However, JavaScript is used to executing code only if a condition is `true`. So if you want a false condition to cause code to execute, you need to switch that false value to true (and any true value to false). That way you can trick JavaScript into executing code after a false condition.

You do this using the NOT operator. This operator reverses the logic of a result; it takes one Boolean value and changes it to the other Boolean value. So it changes `true` to `false` and `false` to `true`. This is sometimes called *negation*.

To use the NOT operator, you put the condition you want reversed in parentheses and put the `!` symbol in front of the parentheses. For example:

```
if (!(degCent < 100))
{
    // Some code
}
```

Any code within the braces will be executed only if the condition `degCent < 100` is false.

The following table details the possible results when using NOT.

Right-Hand Side	Result
true	false
false	true

Multiple Conditions Inside an if Statement

The previous section started by asking how you could use the condition “Is `degFahren` greater than zero but less than 100?” One way of doing this would be to use two `if` statements, one nested inside another. *Nested* simply means that there is an outer `if` statement, and inside this an inner `if` statement. If the condition for the outer `if` statement is `true`, then (and only then) the nested inner `if` statement’s condition will be tested.

Using nested `if` statements, your code would be:

```
if (degCent < 100)
{
    if (degCent > 0)
    {
        document.write("degCent is between 0 and 100");
    }
}
```

This would work, but it's a little verbose and can be quite confusing. JavaScript offers a better alternative — using multiple conditions inside the condition part of the `if` statement. The multiple conditions are strung together with the logical operators you just looked at. So the preceding code could be rewritten like this:

```
if (degCent > 0 && degCent < 100)
{
    document.write("degCent is between 0 and 100");
}
```

The `if` statement's condition first evaluates whether `degCent` is greater than zero. If that is true, the code goes on to evaluate whether `degCent` is less than 100. Only if both of these conditions are true will the `document.write()` code line execute.

Try It Out Multiple Conditions

This example demonstrates multi-condition `if` statements using the `AND`, `OR`, and `NOT` operators. Type the following code, and save it as `ch3_examp2.htm`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<body>

<script type="text/javascript">

var myAge = Number(prompt("Enter your age",30));

if (myAge >= 0 && myAge <= 10)
{
    document.write("myAge is between 0 and 10<br />");
}

if ( !(myAge >= 0 && myAge <= 10) )
{
    document.write("myAge is NOT between 0 and 10<br />");
}

if ( myAge >= 80 || myAge <= 10 )
{
    document.write("myAge is 80 or above OR 10 or below<br />");
}

if ( (myAge >= 30 && myAge <= 39) || (myAge >= 80 && myAge <= 89) )
{
    document.write("myAge is between 30 and 39 or myAge is between 80 and 89");
}

</script>

</body>
</html>
```

Chapter 3: Decisions, Loops, and Functions

When you load it into your browser, a prompt box should appear. Enter the value 30, then press Return, and the lines shown in Figure 3-6 are written to the web page.

myAge is NOT between 0 and 10
myAge is between 30 and 39 or myAge is between 80 and 89

Figure 3-6

The script block starts by defining the variable `myAge` and initializing it to the value entered by the user in the prompt box and converted to a number.

```
var myAge = Number(prompt("Enter your age",30));
```

After this are four `if` statements, each using multiple conditions. You'll look at each in detail in turn.

The easiest way to work out what multiple conditions are doing is to split them up into smaller pieces and then evaluate the combined result. In this example you have entered the value 30, which has been stored in the variable `myAge`. You'll substitute this value into the conditions to see how they work.

Here's the first `if` statement:

```
if (myAge >= 0 && myAge <= 10)
{
    document.write("myAge is between 0 and 10<br />");
}
```

The first `if` statement is asking the question "Is `myAge` between 0 and 10?" You'll take the LHS of the condition first, substituting your particular value for `myAge`. The LHS asks "Is 30 greater than or equal to 0?" The answer is `true`. The question posed by the RHS condition is "Is 30 less than or equal to 10?" The answer is `false`. These two halves of the condition are joined using `&&`, which indicates the AND operator. Using the AND results table shown earlier, you can see that if LHS is `true` and RHS is `false`, you have an overall result of `false`. So the end result of the condition for the `if` statement is `false`, and the code inside the braces won't execute.

Let's move on to the second `if` statement.

```
if ( !(myAge >= 0 && myAge <= 10) )
{
    document.write("myAge is NOT between 0 and 10<br />");
}
```

The second `if` statement is posing the question "Is `myAge` not between 0 and 10?" Its condition is similar to that of the first `if` statement, but with one small difference: You have enclosed the condition inside parentheses and put the NOT operator (!) in front.

The part of the condition inside the parentheses is evaluated and, as before, produces the same result — `false`. However, the NOT operator reverses the result and makes it `true`. Because the `if` statement's condition is `true`, the code inside the braces *will* execute this time, causing a `document.write()` to write a response to the page.

What about the third `if` statement?

```
if ( myAge >= 80 || myAge <= 10 )
{
    document.write("myAge is either 80 and above OR 10 or below<br />");
}
```


The third `if` statement asks, “Is `myAge` greater than or equal to 80, or less than or equal to 10?” Taking the LHS condition first — “Is 30 greater than or equal to 80?” — the answer is `false`. The answer to the RHS condition — “Is 30 less than or equal to 10?” — is again `false`. These two halves of the condition are combined using `||`, which indicates the OR operator. Looking at the OR result table earlier in this section, you see that `false OR false` produces a result of `false`. So again the `if` statement’s condition evaluates to `false`, and the code within the curly braces does not execute.

The final `if` statement is a little more complex.

```
if ( (myAge >= 30 && myAge <= 39) || (myAge >= 80 && myAge <= 89) )
{
    document.write("myAge is between 30 and 39 " +
        "or myAge is between 80 and 89<br />");
}
```

It asks the question, “Is `myAge` between 30 and 39 or between 80 and 89?” Let’s break the condition down into its component parts. There is a left-hand-side and a right-hand-side condition, combined by means of an OR operator. However, the LHS and RHS themselves have an LHS and RHS each, which are combined using AND operators. Notice how parentheses are used to tell JavaScript which parts of the condition to evaluate first, just as you would do with numbers in a mathematical calculation.

Let’s look at the LHS of the condition first, namely `(myAge >= 30 && myAge <= 39)`. By putting the condition into parentheses, you ensure that it’s treated as a single condition; no matter how many conditions are inside the parentheses, it only produces a single result, either `true` or `false`. Breaking down the conditions in the parentheses, you have “Is 30 greater than or equal to 30?” with a result of `true`, and “Is 30 less than or equal to 39?” again with a result of `true`. From the AND table, you know `true AND true` produces a result of `true`.

Now let’s look at the RHS of the condition, namely `(myAge >= 80 && myAge <= 89)`. Again breaking the condition down, you see that the LHS asks, “Is 30 greater than or equal to 80?” which gives a `false` result, and the RHS asks, “Is 30 less than or equal to 89?” which gives a `true` result. You know that `false AND true` gives a `false` result.

Now you can think of your `if` statement’s condition as looking like `(true || false)`. Looking at the OR results table, you can see that `true OR false` gives a result of `true`, so the code within the braces following the `if` statement will execute, and a line will be written to the page.

However, remember that JavaScript does not evaluate conditions where they won’t affect the final result, and the preceding condition is one of those situations. The LHS of the condition evaluated to `true`. After that, it does not matter if the RHS of the condition is `true` or `false` because only one of the conditions in an OR operation needs to be `true` for a result of `true`. Thus JavaScript does not actually evaluate the RHS of the condition. We did so simply for demonstration purposes.

As you have seen, the easiest way to approach understanding or creating multiple conditions is to break them down into the smallest logical chunks. You’ll find that with experience, you will do this almost without thinking, unless you have a particularly tricky condition to evaluate.

Although using multiple conditions is often better than using multiple `if` statements, there are times when it makes your code harder to read and therefore harder to understand and debug. It’s possible to have 10, 20, or more than 100 conditions inside your `if` statement, but can you imagine trying to read an `if` statement with even 10 conditions? If you feel that your multiple conditions are getting too complex, break them down into smaller logical chunks.

Chapter 3: Decisions, Loops, and Functions

For example, imagine you want to execute some code if `myAge` is in the ranges 30–39, 80–89, or 100–115, using different code in each case. You could write the statement like so:

```
if ( (myAge >= 30 && myAge <= 39) || (myAge >= 80 && myAge <= 89) ||
    (myAge >= 100 && myAge <= 115) )
{
    document.write("myAge is between 30 and 39 " +
        "or myAge is between 80 " +
        "and 89 or myAge is between 100 and 115");
}
```

There's nothing wrong with this, but it is starting to get a little long and difficult to read. Instead, you could create another `if` statement for the code executed for the 100–115 range.

else and else if

Imagine a situation where you want some code to execute if a certain condition is true and some other code to execute if it is false. You can achieve this by having two `if` statements, as shown in the following example:

```
if (myAge >= 0 && myAge <= 10)
{
    document.write("myAge is between 0 and 10");
}

if ( !(myAge >= 0 && myAge <= 10) )
{
    document.write("myAge is NOT between 0 and 10");
}
```

The first `if` statement tests whether `myAge` is between 0 and 10, and the second for the situation where `myAge` is not between 0 and 10. However, JavaScript provides an easier way of achieving this: with an `else` statement. Again, the use of the word `else` is similar to its use in the English language. You might say, "If it is raining, I will take an umbrella; otherwise I will take a sun hat." In JavaScript you can say `if` the condition is `true`, then execute one block of code; `else` execute an alternative block. Rewriting the preceding code using this technique, you would have the following:

```
if (myAge >= 0 && myAge <= 10)
{
    document.write("myAge is between 0 and 10");
}
else
{
    document.write("myAge is NOT between 0 and 10");
}
```

Writing the code like this makes it simpler and therefore easier to read. Plus it also saves JavaScript from testing a condition to which you already know the answer.

You could also include another `if` statement with the `else` statement. For example

```
if (myAge >= 0 && myAge <= 10)
{
    document.write("myAge is between 0 and 10");
}
else if ( (myAge >= 30 && myAge <= 39) || (myAge >= 80 && myAge <= 89) )
{
    document.write("myAge is between 30 and 39 " +
        "or myAge is between 80 and 89");
}
else
{
    document.write("myAge is NOT between 0 and 10, " +
        "nor is it between 30 and 39, nor is it between 80 and 89");
}
```

The first `if` statement checks whether `myAge` is between 0 and 10 and executes some code if that's true. If it's false, an `else if` statement checks if `myAge` is between 30 and 39 or 80 and 89, and executes some other code if either of those conditions is true. Failing that, you have a final `else` statement, which catches the situation in which the value of `myAge` did not trigger true in any of the earlier `if` conditions.

When using `if` and `else if`, you need to be extra careful with your curly braces to ensure that the `if` and `else if` statements start and stop where you expect, and you don't end up with an `else` that doesn't belong to the right `if`. This is quite tricky to describe with words — it's easier to see what we mean with an example.

```
if (myAge >= 0 && myAge <= 10)
{
    document.write("myAge is between 0 and 10");
    if (myAge == 5)
    {
        document.write("You're 5 years old");
    }
    else
    {
        document.write("myAge is NOT between 0 and 10");
    }
}
```

Notice that we haven't indented the code. Although this does not matter to JavaScript, it does make the code more difficult for humans to read and hides the missing curly brace that should be before the final `else` statement.

Correctly formatted and with the missing bracket inserted, the code looks like this:

```
if (myAge >= 0 && myAge <= 10)
{
    document.write("myAge is between 0 and 10<br />");
    if (myAge == 5)
    {
        document.write("You're 5 years old");
    }
}
```

Chapter 3: Decisions, Loops, and Functions

```
}
else
{
    document.write("myAge is NOT between 0 and 10");
}
```

As you can see, the code is working now; it is also a lot easier to see which code is part of which `if` block.

Comparing Strings

Up to this point, you have been looking exclusively at using comparison operators with numbers. However, they work just as well with strings. All that's been said and done with numbers applies to strings, but with one important difference. You are now comparing data alphabetically rather than numerically, so there are a few traps to watch out for.

In the following code, you compare the variable `myName`, which contains the string "Paul", with the string literal "Paul".

```
var myName = "Paul";
if (myName == "Paul")
{
    alert("myName is Paul");
}
```

How does JavaScript deal with this? Well, it goes through each letter in turn on the LHS and checks it with the letter in the same position on the RHS to see if it's actually the same. If at any point it finds a difference, it stops, and the result is `false`. If, after having checked each letter in turn all the way to the end, it confirms that they are all the same, it returns `true`. The condition in the preceding `if` statement will return `true`, so you'll see an alert box.

However, string comparison in JavaScript is case sensitive. So "P" is not the same as "p". Taking the preceding example, but changing the variable `myName` to "paul", you find that the condition is `false` and the code inside the `if` statement does not execute.

```
var myName = "paul";
if (myName == "Paul")
{
    alert("myName is Paul");
}
```

The `>=`, `>`, `<=`, and `<` operators work with strings as well as with numbers, but again it is an alphabetical comparison. So "A" < "B" is `true`, because A comes before B in the alphabet. However, JavaScript's case sensitivity comes into play again. "A" < "B" is `true`, but "a" < "B" is `false`. Why? Because uppercase letters are treated as always coming *before* lowercase letters. Why is this? Each letter has a code number in the ASCII and Unicode character sets, and the code numbers for uppercase letters are lower than the code numbers for lowercase letters. This is something to watch out for when writing your own code.

The simplest way to avoid confusion with different cases is to convert both strings to either uppercase or lowercase before you compare them. You can do this easily using the `toUpperCase()` or `toLowerCase()` function, which you'll learn about in Chapter 4.

The switch Statement

You saw earlier how the `if` and `else if` statements could be used for checking various conditions; if the first condition is not valid, then another is checked, and another, and so on. However, when you want to check the value of a particular variable for a large number of possible values, there is a more efficient alternative, namely the `switch` statement. The structure of the `switch` statement is given in Figure 3-7.

The best way to think of the `switch` statement is “Switch to the code where the case matches.” The `switch` statement has four important elements:

- ❑ The test expression
- ❑ The case statements
- ❑ The break statements
- ❑ The default statement

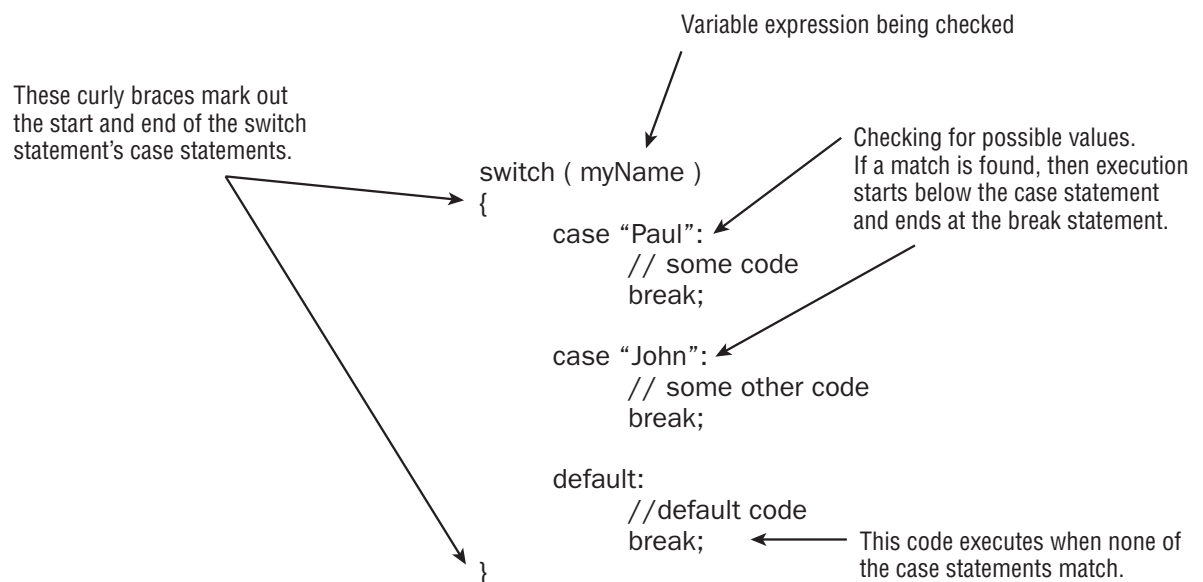


Figure 3-7

The test expression is given in the parentheses following the `switch` keyword. In the previous example, you are testing using the variable `myName`. Inside the parentheses, however, you could have any valid expression.

Next come the case statements. The case statements do the condition checking. To indicate which case statements belong to your `switch` statement, you must put them inside the curly braces following the test expression. Each case statement specifies a value, for example `"Paul"`. The case statement then acts like `if (myName == "Paul")`. If the variable `myName` did contain the value `"Paul"`, execution would commence from the code starting below the `case "Paul"` statement and would continue to the end of the `switch` statement. This example has only two case statements, but you can have as many as you like.

In most cases, you want only the block of code directly underneath the relevant case statement to execute, not *all* the code below the relevant case statement, including any other case statements.

Chapter 3: Decisions, Loops, and Functions

To achieve this, you put a `break` statement at the end of the code that you want executed. This tells JavaScript to stop executing at that point and leave the `switch` statement.

Finally you have the `default` case, which (as the name suggests) is the code that will execute when none of the other `case` statements match. The `default` statement is optional; if you have no default code that you want to execute, you can leave it out, but remember that in this case no code will execute if no `case` statements match. It is a good idea to include a `default` case, unless you are absolutely sure that you have all your options covered.

Try It Out Using the `switch` Statement

Let's take a look at the `switch` statement in action. The following example illustrates a simple guessing game. Type the code and save it as `ch3_examp3.htm`.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<body>

<script type="text/javascript">

var secretNumber = prompt("Pick a number between 1 and 5:", "");
secretNumber = parseInt(secretNumber);

switch (secretNumber)
{
case 1:
    document.write("Too low!");
    break;

case 2:
    document.write("Too low!");
    break;

case 3:
    document.write("You guessed the secret number!");
    break;

case 4:
    document.write("Too high!");
    break;

case 5:
    document.write("Too high!");
    break;

default:
    document.write("You did not enter a number between 1 and 5.");
    break;
}
```

```
document.write("<br />Execution continues here");

</script>

</body>
</html>
```

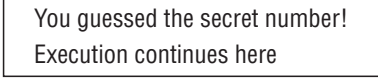
Load this into your browser and enter, for example, the value 1 in the prompt box. You should then see something like what is shown in Figure 3-8.



Too low!
Execution continues here

Figure 3-8

If, on the other hand, you enter the value 3, you should see a friendly message letting you know that you guessed the secret number correctly, as shown in Figure 3-9.



You guessed the secret number!
Execution continues here

Figure 3-9

First you declare the variable `secretNumber` and set it to the value entered by the user via the prompt box. Note that you use the `parseInt()` function to convert the string that is returned from `prompt()` to an integer value.

```
var secretNumber = prompt("Pick a number between 1 and 5:", "");
secretNumber = parseInt(secretNumber);
```

Next you create the start of the `switch` statement.

```
switch (secretNumber)
{
```

The expression in parentheses is simply the variable `secretNumber`, and it's this number that the `case` statements will be compared against.

You specify the block of code encompassing the `case` statements using curly braces. Each `case` statement checks one of the numbers between 1 and 5, because this is what you have specified to the user that she should enter. The first simply outputs a message that the number she has entered is too low.

```
case 1:
    document.write("Too low!");
    break;
```

The second `case` statement, for the value 2, has the same message, so the code is not repeated here. The third `case` statement lets the user know that she has guessed correctly.

```
case 3:
    document.write("You guessed the secret number!");
    break;
```

Chapter 3: Decisions, Loops, and Functions

Finally, the fourth and fifth `case` statements output a message that the number the user has entered is too high.

```
case 4:
    document.write("Too high!");
    break;
```

You do need to add a default case in this example, since the user might very well (despite the instructions) enter a number that is not between 1 and 5, or even perhaps a letter. In this case, you add a message to let the user know that there is a problem.

```
default:
    document.write("You did not enter a number between 1 and 5.");
    break;
```

A default statement is also very useful for picking up bugs — if you have coded some of the `case` statements incorrectly, you will pick that up very quickly if you see the `default` code being run when it shouldn't be.

You finally have added the closing brace indicating the end of the `switch` statement. After this you output a line to indicate where the execution continues.

```
}
document.write("<br />Execution continues here");
```

Note that each `case` statement ends with a `break` statement. This is important to ensure that execution of the code moves to the line after the end of the `switch` statement. If you forget to include this, you could end up executing the code for each case following the case that matches.

Executing the Same Code for Different Cases

You may have spotted a problem with the `switch` statement in this example — you want to execute the same code if the user enters a 1 or a 2, and the same code for a 4 or a 5. However, in order to achieve this, you have had to repeat the code in each case. What you want is an easier way of getting JavaScript to execute the same code for different cases. Well, that's easy! Simply change the code so that it looks like this:

```
switch (secretNumber)
{
    case 1:
    case 2:
        document.write("Too low!");
        break;

    case 3:
        document.write("You guessed the secret number!");
        break;

    case 4:
    case 5:
```



```
document.write("Too high!");  
break;  
  
default:  
    document.write("You did not enter a number between 1 and 5.");  
    break;  
}
```

If you load this into your browser and experiment with entering some different numbers, you should see that it behaves exactly like the previous code.

Here, you are making use of the fact that if there is no `break` statement underneath the code for a certain case statement, execution will continue through each following case statement until a `break` statement or the end of the `switch` is reached. Think of it as a sort of free fall through the `switch` statement until you hit the `break` statement.

If the case statement for the value 1 is matched, execution simply continues until the `break` statement under case 2, so effectively you can execute the same code for both cases. The same technique is used for the case statements with values 4 and 5.

Looping — The `for` and `while` Statements

Looping means repeating a block of code when a condition is `true`. This is achieved in JavaScript with the use of two statements, the `while` statement and the `for` statement. You'll be looking at these shortly, but why would you want to repeat blocks of code anyway?

Well, take the situation where you have a series of results, say the average temperature for each month in a year, and you want to plot these on a graph. The code needed for plotting each point will most likely be the same. So, rather than write the code 12 times (once for each point), it's much easier to execute the same code 12 times by using the next item of data in the series. This is where the `for` statement would come in handy, because you know how many times you want the code to execute.

In another situation, you might want to repeat the same piece of code when a certain condition is `true`, for example, while the user keeps clicking a Start Again button. In this situation, the `while` statement would be very useful.

The `for` Loop

The `for` statement enables you to repeat a block of code a certain number of times. The syntax is illustrated in Figure 3-10.

Let's look at the makeup of a `for` statement. You can see from Figure 3-10 that, just like the `if` and `switch` statements, the `for` statement also has its logic inside parentheses. However, this time that logic split into three parts, each part separated by a semicolon. For example, in Figure 3-10 you have the following:

```
(var loopCounter = 1; loopCounter <= 3; loopCounter++)
```

Chapter 3: Decisions, Loops, and Functions

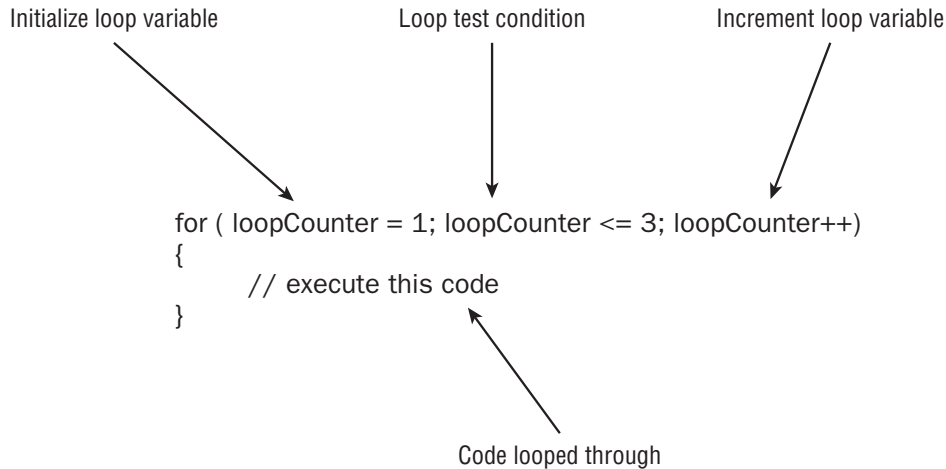


Figure 3-10

The first part of the `for` statement's logic is the *initialization* part of the `for` statement. To keep track of how many times you have looped through the code, you need a variable to keep count. It's in the initialization part that you initialize variables. In the example you have declared `loopCounter` and set it to the value of 1. This part is only executed once during the execution of the loops, unlike the other parts. You don't need to declare the variable if it was declared earlier in the code.

```
var loopCounter;  
for (loopCounter = 1; loopCounter <= 3; loopCounter++)
```

Following the semicolon, you have the *test condition* part of the `for` statement. The code inside the `for` statement will keep executing for as long as this test condition evaluates to `true`. After the code is looped through each time, this condition is tested. In Figure 3-10, you execute for as long as `loopCounter` is less than or equal to 3. The number of times a loop is performed is often called the number of *iterations*.

Finally, you have the *increment* part of the `for` loop, where variables in our loop's test condition have their values incremented. Here you can see that `loopCounter` is incremented by one by means of the `++` operator you saw in Chapter 2. Again, this part of the `for` statement is repeated with every loop of the code. Although we call it the increment part, it can actually be used to decrease or *decrement* the value — for example, if you wanted to count down from the top element in an array to the first.

After the `for` statement comes the block of code that will be executed repeatedly, as long as the test condition is `true`. This block of code is contained within curly braces. If the condition is never `true`, even at the first test of the loop condition, then the code inside the `for` loop will be skipped over and never executed.

Putting all this together, how does the `for` loop work?

1. Execute initialization part of the `for` statement.
2. Check the test condition. If `true`, continue; if not, exit the `for` statement.

3. Execute code in the block after the `for` statement.
4. Execute the increment part of the `for` statement.
5. Repeat steps 2 through 4 until the test condition is `false`.

Try It Out Converting a Series of Fahrenheit Values

Let's change the temperature converter so that it converts a series of values, stored in an array, from Fahrenheit to centigrade. You will be using the `for` statement to go through each element of the array. Type the code and save it as `ch3_examp4.htm`.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<body>

<script type="text/javascript">

var degFahren = new Array(212, 32, -459.15);
var degCent = new Array();
var loopCounter;

for (loopCounter = 0; loopCounter <= 2; loopCounter++)
{
    degCent[loopCounter] = 5/9 * (degFahren[loopCounter] - 32);
}

for (loopCounter = 2; loopCounter >= 0; loopCounter--)
{
    document.write("Value " + loopCounter + " was " + degFahren[loopCounter] +
        " degrees Fahrenheit");
    document.write(" which is " + degCent[loopCounter] +
        " degrees centigrade<br />");
}

</script>

</body>
</html>
```

On loading this into your browser, you'll see a series of three lines in the page, containing the results of converting our array of Fahrenheit values into centigrade (as shown in Figure 3-11).

Value 2 was -459.15 degrees Fahrenheit which is -272.8611111111111 degrees centigrade
Value 1 was 32 degrees Fahrenheit which is 0 degrees centigrade
Value 0 was 212 degrees Fahrenheit which is 100 degrees centigrade

Figure 3-11

The first task is to declare the variables you are going to use. First, you declare and initialize `degFahren` to contain an array of three values: 212, 32, and -459.15. Next, `degCent` is declared as an empty array.

Chapter 3: Decisions, Loops, and Functions

Finally, `loopCounter` is declared and will be used to keep track of which array index you are accessing during your looping.

```
var degFahren = new Array(212, 32, -459.15);
var degCent = new Array();
var loopCounter;
```

Following this comes our first `for` loop.

```
for (loopCounter = 0; loopCounter <= 2; loopCounter++)
{
    degCent[loopCounter] = 5/9 * (degFahren[loopCounter] - 32);
}
```

In the first line, you start by initializing the `loopCounter` to 0. Then the `for` loop's test condition, `loopCounter <= 2`, is checked. If this condition is `true`, the loop executes for the first time. After the code inside the curly braces has executed, the incrementing part of the `for` loop, `loopCounter++`, will be executed, and then the test condition will be re-evaluated. If it's still `true`, another execution of the loop code is performed. This continues until the `for` loop's test condition evaluates to `false`, at which point looping will end, and the first statement after the closing curly brace will be executed.

The code inside the curly braces is the equation you saw in earlier examples, only this time you are placing its result into the `degCent` array, with the index being the value of `loopCounter`.

In the second `for` loop, you write the results contained in the `degCent` array to the screen.

```
for (loopCounter = 2; loopCounter >= 0; loopCounter--)
{
    document.write("Value " + loopCounter + " was " + degFahren[loopCounter] +
        " degrees Fahrenheit");
    document.write(" which is " + degCent[loopCounter] +
        " degrees centigrade<br />");
}
```

This time you're counting *down* from 2 to 0. The variable `loopCounter` is initialized to 2, and the loop condition remains `true` until `loopCounter` is less than 0. This time `loopCounter` is actually decremented each time rather than incremented, by means of `loopCounter--`. Again, `loopCounter` is serving a dual purpose: It keeps count of how many loops you have done and also provides the index position in the array.

Note that in these examples, you've used whole numbers in your loops. However, there is no reason why you can't use fractional numbers, although it's much less common to do so.

The `for...in` Loop

This loop works primarily with arrays, and as you'll see in the next chapter, it also works with something called objects. It enables you to loop through each element in the array without having to know how many elements the array actually contains. In plain English, what this loop says is "For each element in the array, execute some code." Rather than having to work out the index number of each element, the `for...in` loop does it for you and automatically moves to the next index with each iteration (loop through).

Its syntax for use with arrays is:

```
for (index in arrayName)
{
    //some code
}
```

In this code extract, `index` is a variable you declare prior to the loop, which will automatically be populated with the next index value in the array. `arrayName` is the name of the variable holding the array you want to loop through.

Let's look at an example to make things clearer. You'll define an array and initialize it with three values.

```
var myArray = new Array("Paul", "Paula", "Pauline");
```

To access each element using a conventional `for` loop, you'd write this:

```
var loopCounter;
for (loopCounter = 0; loopCounter < 3; loopCounter++)
{
    document.write(myArray[loopCounter]);
}
```

To do exactly the same thing with the `for...in` loop, you write this:

```
var elementIndex;
for (elementIndex in myArray)
{
    document.write(myArray[elementIndex]);
}
```

As you can see, the code in the second example is a little clearer, as well as shorter. Both methods work equally well and will iterate three times. However, if you increase the size of the array, for example, by adding the element `myArray[3] = "Philip"`, the first method will still loop only through the first three elements in the array, whereas the second method will loop through all four elements.

The while Loop

Whereas the `for` loop is used for looping a certain number of times, the `while` loop enables you to test a condition and keep on looping while it's true. The `for` loop is useful when you know how many times you need to loop, for example when you are looping through an array that you know has a certain number of elements. The `while` loop is more useful when you don't know how many times you'll need to loop. For example, if you are looping through an array of temperature values and want to continue looping when the temperature value contained in the array element is less than 100, you will need to use the `while` statement.

Let's take a look at the structure of the `while` statement, as illustrated in Figure 3-12.

Chapter 3: Decisions, Loops, and Functions

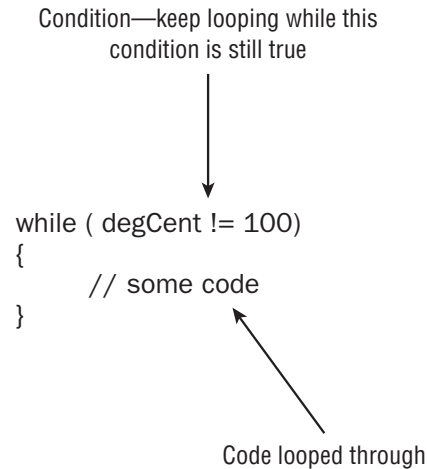


Figure 3-12

You can see that the `while` loop has fewer parts to it than the `for` loop. The `while` loop consists of a condition which, if it evaluates to `true`, causes the block of code inside the curly braces to execute once; then the condition is re-evaluated. If it's still `true`, the code is executed again, the condition is re-evaluated, and so on until the condition evaluates to `false`.

One thing to watch out for is that if the condition is `false` to start with, the `while` loop never executes. For example:

```
degCent = 100;  
  
while (degCent != 100)  
{  
    // some code  
}
```

Here, the loop will run if `degCent` does not equal 100. However, since `degCent` is 100, the condition is `false`, and the code never executes.

In practice you would normally expect the loop to execute once; whether it executes again will depend on what the code inside the loop has done to variables involved in the loop condition. For example:

```
degCent = new Array();  
degFahren = new Array(34, 123, 212);  
var loopCounter = 0;  
while (loopCounter < 3)  
{  
    degCent[loopCounter] = 5/9 * (degFahren[loopCounter] - 32);  
    loopCounter++;  
}
```

The loop will execute so long as `loopCounter` is less than 3. It's the code inside the loop (`loopCounter++;`) that increments `loopCounter` and will eventually cause `loopCounter < 3` to be `false` so that the loop stops. Execution will then continue on the first line after the closing brace of the `while` statement.

Something to watch out for is the *infinite loop* — a loop that will never end. Suppose you forgot to include the `loopCounter++;` line in the code. Leaving this line out would mean that `loopCounter` will remain at 0, so the condition `(loopCounter < 3)` will always be `true`, and the loop will continue until the user gets bored and cross, and shuts down her browser. However, it is an easy mistake to make and one JavaScript won't warn you about.

It's not just missing lines that can cause infinite loops but also mistakes inside the loop's code. For example:

```
var testVariable = 0;
while (testVariable <= 10)
{
    alert("Test Variable is " + testVariable);
    testVariable++;
    if (testVariable = 10)
    {
        alert("The last loop");
    }
}
```

See if you can spot the deliberate mistake that leads to an infinite loop — yes, it's the `if` statement that will cause this code to go on forever. Instead of using `==` as the comparison operator in the condition of the `if` statement, you put `=`, so `testVariable` is set to 10 again in each loop, despite the line `testVariable++`. This means that at the start of each loop, the test condition always evaluates to `true`, since 10 is less than or equal to 10. Put the extra `=` in to make `if (testVariable == 10)`, and everything is fine.

The *do...while* loop

With the `while` loop, you saw that the code inside the loop only executes if the condition is `true`; if it's `false`, the code never executes, and execution instead moves to the first line after the `while` loop. However, there may be times when you want the code in the `while` loop to execute at least once, regardless of whether the condition in the `while` statement evaluates to `true`. It might even be that some code inside the `while` loop needs to be executed before you can test the `while` statement's condition. It's situations like this for which the `do...while` loop is ideal.

Look at an example in which you want to get the user's age via a prompt box. You want to show the prompt box but also make sure that what the user has entered is a number.

```
var userAge;
do
{
    userAge = prompt("Please enter your age", "");
}
while (isNaN(userAge) == true);
```

The code line within the loop —

```
userAge = prompt("Please enter your age", "");
```

— will be executed regardless of the `while` statement's condition. This is because the condition is not checked *until* one loop has been executed. If the condition is `true`, the code is looped through again. If it's `false`, looping stops.

Chapter 3: Decisions, Loops, and Functions

Note that within the `while` statement's condition, you are using the `isNaN()` function that you saw in Chapter 2. This checks whether the `userAge` variable's value is `NaN` (not a number). If it is not a number, the condition returns a value of `true`; otherwise it returns `false`. As you can see from the example, it enables you to test the user input to ensure the right data has been entered. The user might lie about his age, but at least you know he entered a number!

The `do...while` loop is fairly rare; there's not much you can't do without it, so it's best avoided unless really necessary.

The break and continue Statements

You met the `break` statement earlier when you looked at the `switch` statement. Its function inside a `switch` statement is to stop code execution and move execution to the next line of code after the closing curly brace of the `switch` statement. However, the `break` statement can also be used as part of the `for` and `while` loops when you want to exit the loop prematurely. For example, suppose you're looping through an array, as you did in the temperature conversion example, and you hit an invalid value. In this situation, you might want to stop the code in its tracks, notify the user that the data is invalid, and leave the loop. This is one situation where the `break` statement comes in handy.

Let's see how you could change the example where you converted a series of Fahrenheit values (`ch3_examp4.htm`) so that if you hit a value that's not a number you stop the loop and let the user know about the invalid data.

```
<script language="JavaScript" type="text/javascript">
var degFahren = new Array(212, "string data", -459.67);
var degCent = new Array();
var loopCounter;

for (loopCounter = 0; loopCounter <= 2; loopCounter++)
{
    if (isNaN(degFahren[loopCounter]))
    {
        alert("Data '" + degFahren[loopCounter] + "' at array index " +
            loopCounter + " is invalid");
        break;
    }

    degCent[loopCounter] = 5/9 * (degFahren[loopCounter] - 32);
}
```

You have changed the initialization of the `degFahren` array so that it now contains some invalid data. Then, inside the `for` loop, an `if` statement is added to check whether the data in the `degFahren` array is not a number. This is done by means of the `isNaN()` function; it returns `true` if the value passed to it in the parentheses, here `degFahren[loopCounter]`, is not a number. If the value is not a number, you tell the user where in the array you have the invalid data. Then you break out of the `for` loop altogether, using the `break` statement, and code execution continues on the first line after the end of the `for` statement.

That's the `break` statement, but what about `continue`? The `continue` statement is similar to `break` in that it stops the execution of a loop at the point where it is found, but instead of leaving the loop, it starts

execution at the next iteration, starting with the `for` or `while` statement's condition being re-evaluated, just as if the last line of the loop's code had been reached.

In the `break` example, it was all or nothing — if even one piece of data was invalid, you broke out of the loop. It might be better if you tried to convert all the values in `degFahren`, but if you hit an invalid item of data in the array, you notify the user and continue with the next item, rather than giving up as our `break` statement example does.

```
if (isNaN(degFahren[loopCounter]))
{
    alert("Data '" + degFahren[loopCounter] + "' at array index " +
        loopCounter + " is invalid");
    continue;
}
```

Just change the `break` statement to a `continue`. You will still get a message about the invalid data, but the third value will also be converted.

Functions

A function is something that performs a particular task. Take a pocket calculator as an example. It performs lots of basic calculations, such as addition and subtraction. However, many also have function keys that perform more complex operations. For example, some calculators have a button for calculating the square root of a number, and others even provide statistical functions, such as the calculation of an average. Most of these functions could be done with the basic mathematical operations of add, subtract, multiply, and divide, but that might take a lot of steps — it's much simpler for the user if she only needs to press one button. All she needs to do is provide the data — numbers in this case — and the function key does the rest.

Functions in JavaScript work a little like the function buttons on a pocket calculator: They encapsulate a block of code that performs a certain task. Over the course of the book so far, you have come across a number of handy built-in functions that perform a certain task, such as the `parseInt()` and `parseFloat()` functions, which convert strings to numbers, and the `isNaN()` function, which tells you whether a particular value can be converted to a number. Some of these functions return data, such as `parseInt()`, which returns an integer number; others simply perform an action but return no data. You'll also notice that some functions can be passed data, whereas others cannot. For example, the `isNaN()` function needs to be passed some data, which it checks to see if it is `NaN`. The data that a function requires to be passed are known as its *parameter(s)*.

As you work your way through the book, you'll be coming across many more useful built-in functions, but wouldn't it be great to be able to write your own functions? After you've worked out, written, and debugged a block of code to perform a certain task, it would be nice to be able to call it again and again when you need it. JavaScript gives us the ability to do just that, and this is what you'll be concentrating on in this section.

Creating Your Own Functions

Creating and using your own functions is very simple. Figure 3-13 shows an example of a function.

Chapter 3: Decisions, Loops, and Functions

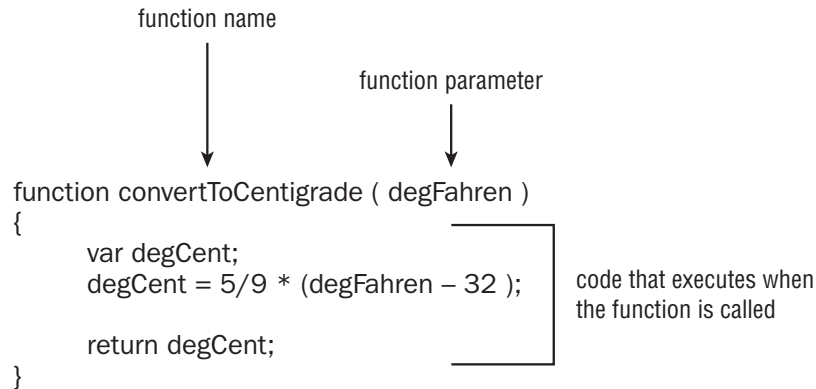


Figure 3-13

You’ve probably already realized what this function does and how the code works. Yes, it’s the infamous Fahrenheit-to-centigrade conversion code again.

Each function you define in JavaScript must be given a unique name for that particular page. The name comes immediately after the `function` keyword. To make life easier for yourself, try using meaningful names so that when you see it being used later in your code, you’ll know exactly what it does. For example, a function that takes as its parameters someone’s birthday and today’s date and returns the person’s age could be called `getAge()`. However, the names you can use are limited, much as variable names are. For example, you can’t use words reserved by JavaScript, so you can’t call your function `with()` or `while()`.

The parameters for the function are given in parentheses after the function’s name. A parameter is just an item of data that the function needs to be given in order to do its job. Usually, not passing the required parameters will result in an error. A function can have zero or more parameters, though even if it has no parameters, you must still put the open and close parentheses after its name. For example, the top of your function definition must look like the following:

```
function myNoParamFunction()
```

You then write the code, which the function will execute when called on to do so. All the function code must be put in a block with a pair of curly braces.

Functions also give you the ability to return a value from a function to the code that called it. You use the `return` statement to return a value. In the example function given earlier, you return the value of the variable `degCent`, which you have just calculated. You don’t have to return a value if you don’t want to, but you should always include a `return` statement at the end of your function, although JavaScript is a very forgiving language and won’t have a problem if you don’t use a `return` statement at all.

When JavaScript comes across a `return` statement in a function, it treats it a bit like a `break` statement in a `for` loop — it exits the function, returning any value specified after the `return` keyword.

You’ll probably find it useful to build up a “library” of functions that you use frequently in JavaScript code, which you can cut and paste into your page whenever you need them.

Having created your functions, how do you use them? Unlike the code you’ve seen so far, which executes when JavaScript reaches that line, functions only execute if you ask them to, which is termed *calling* or

Chapter 3: Decisions, Loops, and Functions

invoking the function. You call a function by writing its name at the point where you want it to be called and making sure that you pass any parameters it needs, separated by commas. For example:

```
myTemp = convertToCentigrade(212);
```

This line calls the `convertToCentigrade()` function you saw earlier, passing 212 as the parameter and storing the return value from the function (that is, 100) in the `myTemp` variable.

Have a go at creating your own functions now, taking a closer look at how parameters are passed. Parameter passing can be a bit confusing, so you'll first create a simple function that takes just one parameter (the user's name) and writes it to the page in a friendly welcome string. First, you need to think of a name for your function. A short but descriptive name is `writeUserWelcome()`. Now you need to define what parameters the function expects to be passed. There's only one parameter — the user name. Defining parameters is a little like defining variables — you need to stick to the same rules for naming, so that means no spaces, special characters, or reserved words. Let's call your parameter `userName`. You need to add it inside parentheses to the end of the function name (note that you don't put a semicolon at the end of the line).

```
function writeUserWelcome(userName)
```

Okay, now you have defined your function name and its parameters; all that's left is to create the function body — that is, the code that will be executed when the function is called. You mark out this part of the function by wrapping it in curly braces.

```
function writeUserWelcome(userName)
{
    document.write("Welcome to my website " + userName + "<br />");
    document.write("Hope you enjoy it!");
}
```

The code is simple enough; you write out a message to the web page using `document.write()`. You can see that `userName` is used just as you'd use any normal variable; in fact, it's best to think of parameters as normal variables. The value that the parameter has will be that specified by the JavaScript code where the function was called.

Let's see how you would call this function.

```
writeUserWelcome("Paul");
```

Simple, really — just write the name of the function you want to call, and then in parentheses add the data to be passed to each of the parameters, here just one piece. When the code in the function is executed, the variable `userName`, used in the body of the function code, will contain the text "Paul".

Suppose you wanted to pass two parameters to your function — what would you need to change? Well, first you'd have to alter the function definition. Imagine that the second parameter will hold the user's age — you could call it `userAge` since that makes it pretty clear what the parameter's data represents. Here is the new code:

```
function writeUserWelcome(userName, userAge)
{
    document.write("Welcome to my website" + userName + "<br />");
```

Chapter 3: Decisions, Loops, and Functions

```
document.write("Hope you enjoy it<br />");
document.write("Your age is " + userAge);
}
```

You've added a line to the body of the function that uses the parameter you have added. To call the function, you'd write the following:

```
writeUserWelcome("Paul", 31);
```

The second parameter is a number, so there is no need for quotes around it. Here the `userName` parameter will be `Paul`, and the second parameter, `userAge`, will be `31`.

Try It Out Fahrenheit to Centigrade Function

Let's rewrite the temperature converter page using functions. You can cut and paste most of this code from `ch3_examp4.htm` — the parts that have changed have been highlighted. When you've finished, save it as `ch3_examp5.htm`.

```
<html>
<body>

<script language="JavaScript" type="text/javascript">

function convertToCentigrade(degFahren)
{
    var degCent;
    degCent = 5/9 * (degFahren - 32);

    return degCent;
}

var degFahren = new Array(212, 32, -459.15);
var degCent = new Array();
var loopCounter;

for (loopCounter = 0; loopCounter <= 2; loopCounter++)
{
    degCent[loopCounter] = convertToCentigrade(degFahren[loopCounter]);
}

for (loopCounter = 2; loopCounter >= 0; loopCounter--)
{
    document.write("Value " + loopCounter + " was " + degFahren[loopCounter] +
        " degrees Fahrenheit");
    document.write(" which is " + degCent[loopCounter] +
        " degrees centigrade<br />");
}

</script>

</body>
</html>
```

When you load this page into your browser, you should see exactly the same results that you had with `ch3_examp4.htm`.

At the top of the script block you declare your `convertToCentigrade()` function. You saw this function earlier:

```
function convertToCentigrade(degFahren)
{
    var degCent;
    degCent = 5/9 * (degFahren - 32);

    return degCent;
}
```

If you're using a number of separate script blocks in a page, it's very important that the function be defined before any script calls it. If you have a number of functions, you may want to put them all in their own script block at the top of the page — between the `<head>` and `</head>` tags is good. That way you know where to find all your functions, and you can be sure that they have been declared before they have been used.

You should be pretty familiar with how the code in the function works. You declare a variable `degCent`, do your calculation, store its result in `degCent`, and then return `degCent` back to the calling code. The function's parameter is `degFahren`, which provides the information the calculation needs.

Following the function declaration is the code that executes when the page loads. First you define the variables you need, and then you have the two loops that calculate and then output the results. This is mostly the same as before, apart from the first `for` loop.

```
for (loopCounter = 0; loopCounter <= 2; loopCounter++)
{
    degCent[loopCounter] = convertToCentigrade(degFahren[loopCounter]);
}
```

The code inside the first `for` loop puts the value returned by the function `convertToCentigrade()` into the `degCent` array.

There is a subtle point to the code in this example. Notice that you declare the variable `degCent` within your function `convertToCentigrade()`, and you also declare it as an array after the function definition.

Surely this isn't allowed?

Well, this leads neatly to the next topic of this chapter — variable scope.

Variable Scope and Lifetime

What is meant by *scope*? Well, put simply, it's the scope or extent of a variable's availability — which parts of your code can access a variable and the data it contains. Any variables declared in a web page outside of a function will be available to all script on the page, whether that script is inside a function or otherwise — we term this a *global* or *page-level scope*. However, variables declared inside a function are

Chapter 3: Decisions, Loops, and Functions

visible *only* inside that function — no code outside the function can access them. So, for example, you could declare a variable `degCent` in every function you have on a page *and* once on the page outside any function. However, you can't declare the variable *more* than once inside any one function or *more* than once on the page outside the functions. Note that reusing a variable name throughout a page in this way, although not illegal, is not standard good practice — it can make the code very confusing to read.

Function parameters are similar to variables: They can't be seen outside the function, and although you can declare a variable in a function with the same name as one of its parameters, it would cause a lot of confusion and might easily lead to subtle bugs being overlooked. It's therefore bad coding practice and best avoided, if only for the sake of your sanity when it comes to debugging!

So what happens when the code inside a function ends and execution returns to the point at which the code was called? Do the variables defined within the function retain their value when you call the function the next time?

The answer is no: Variables not only have the scope property — where they are visible — but they also have a *lifetime*. When the function finishes executing, the variables in that function die and their values are lost, unless you return one of them to the calling code. Every so often JavaScript performs garbage collection (which we talked about in Chapter 2), whereby it scans through the code and sees if any variables are no longer in use; if so, the data they hold are freed from memory to make way for the data of other variables.

Given that global variables can be used anywhere, why not make all of them global? Global variables are great when you need to keep track of data on a global basis. However, because they are available for modification anywhere in your code, it does mean that if they are changed incorrectly due to a bug, that bug could be anywhere within the code, making debugging difficult. It's best, therefore, to keep global variable use to a minimum, though sometimes they are a necessary evil — for example, when you need to share data among different functions.

Summary

In this chapter you have concluded your look at the core of the JavaScript language and its syntax. Everything from now on builds on these foundations, and with the less interesting syntax under your belt, you can move on to more interesting things in the remainder of the book.

The chapter looked at the following:

- ❑ **Decision making with the `if` and `switch` statements.** The ability to make decisions is essentially what gives the code its “intelligence.” Based on whether a condition is `true` or `false`, you can decide on a course of action to follow.
- ❑ **Comparison operators.** The comparison operators compare the value on the left of the operator (left-hand side, LHS) with the value on the right of the operator (right-hand side, RHS) and return a Boolean value. Here is a list of the main comparison operators:
 - ❑ `==` is the LHS equal to the RHS?
 - ❑ `!=` is the LHS not equal to the RHS?
 - ❑ `<=` is the LHS less than or equal to the RHS?

- ☐ `>=` is the LHS greater than or equal to the RHS?
- ☐ `<` is the LHS less than the RHS?
- ☐ `>` is the LHS greater than the RHS?
- ☐ **The `if` statement.** Using the `if` statement, you can choose to execute a block of code (defined by being in curly braces) when a condition is `true`. The `if` statement has a test condition, specified in parentheses. If this condition evaluates to `true`, the code after the `if` statement will execute.
- ☐ **The `else` statement.** If you want code to execute when the `if` statement is `false`, you can use the `else` statement that appears after the `if` statement.
- ☐ **Logical operators.** To combine conditions, you can use the three logical operators: AND, OR, and NOT, represented by `&&`, `||`, and `!`, respectively.
 - ☐ The AND operator returns `true` only if both sides of the expression are `true`.
 - ☐ The OR operator returns `true` when either one or both sides of an expression are `true`.
 - ☐ The NOT operator reverses the logic of an expression.
- ☐ **The `switch` statement.** This compares the result of an expression with a series of possible cases and is similar in effect to a multiple `if` statement.
- ☐ **Looping with `for`, `for...in`, `while`, and `do...while`.** It's often necessary to repeat a block of code a number of times, something JavaScript enables by looping.
 - ☐ **The `for` loop.** Useful for looping through code a certain number of times, the `for` loop consists of three parts: the initialization, test condition, and increment parts. Looping continues while the test condition is `true`. Each loop executes the block of code and then executes the increment part of the `for` loop before re-evaluating the test condition to see if the results of incrementing have changed it.
 - ☐ **The `for...in` loop.** This is useful when you want to loop through an array without knowing the number of elements in the array. JavaScript works this out for you so that no elements are missed.
 - ☐ **The `while` loop.** This is useful for looping through some code for as long as a test condition remains `true`. It consists of a test condition and the block of code that's executed only if the condition is `true`. If the condition is never `true`, the code never executes.
 - ☐ **The `do...while` loop.** This is similar to a `while` loop, except that it executes the code once and then keeps executing the code as long as the test condition remains `true`.
 - ☐ **`break` and `continue` statements.** Sometimes you have a good reason to break out of a loop prematurely, in which case you need to use the `break` statement. On hitting a `break` statement, code execution stops for the block of code marked out by the curly braces and starts immediately after the closing brace. The `continue` statement is similar to `break`, except that when code execution stops at that point in the loop, the loop is not broken out of but instead continues as if the end of that reiteration had been reached.
- ☐ **Functions are reusable bits of code.** JavaScript has a lot of built-in functions that provide programmers services, such as converting a string to a number. However, JavaScript also enables you to define and use your own functions using the `function` keyword. Functions can have zero or more parameters passed to them and can return a value if you so wish.

Chapter 3: Decisions, Loops, and Functions

- ❑ **Variable scope and lifetime.** Variables declared outside a function are available globally — that is, anywhere in the page. Any variables defined inside a function are private to that function and can't be accessed outside of it. Variables have a lifetime, the length of which depends on where the variable was declared. If it's a global variable, its lifetime is that of the page — while the page is loaded in the browser, the variable remains alive. For variables defined in a function, the lifetime is limited to the execution of that function. When the function has finished being executed, the variables die, and their values are lost. If the function is called again later in the code, the variables will be empty.

Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

1. A junior programmer comes to you with some code that appears not to work. Can you spot where he went wrong? Give him a hand and correct the mistakes.

```
var userAge = prompt("Please enter your age");

if (userAge = 0);
{
    alert("So you're a baby!");
}
else if ( userAge < 0 | userAge > 200)
    alert("I think you may be lying about your age");
else
{
    alert("That's a good age");
}
```

2. Using `document.write()`, write code that displays the results of the 12 times table. Its output should be the results of the calculations.

```
12 * 1 = 12
12 * 2 = 24
12 * 3 = 36
...
12 * 11 = 132
12 * 12 = 144
```

3. Change the code of Question 2 so that it's a function that takes as parameters the times table required and the values at which it should start and end. For example, you might try the four times table displayed starting with $4 * 4$ and ending at $4 * 9$.
4. Modify the code of Question 3 to request the times table to be displayed from the user; the code should continue to request and display times tables until the user enters **-1**. Additionally, do a check to make sure that the user is entering a valid number; if the number is not valid, ask the user to re-enter it.

4

Common Mistakes, Debugging, and Error Handling

Even a JavaScript guru makes mistakes, even if they are just annoying typos. In particular, when code expands to hundreds of lines, the chance of something going wrong becomes much greater. In proportion, the difficulty in finding these mistakes, or bugs, also increases. In this chapter you will look at various techniques that will help you minimize the problems that arise from this situation.

You'll start by taking a look at the top seven JavaScript coding mistakes. After you know what they are, you'll be able to look out for them when writing code, hopefully, so that you won't make them so often!

Then you'll look at how you can cope with errors when they do happen, so that you prevent users from seeing your coding mistakes.

Finally, you'll look at the debugging tools in Microsoft's Internet Explorer (IE8), Firebug (an add-on for Firefox), Safari's and Chrome's Web Inspector, and Opera's Dragonfly. You'll see how you can use these tools to step through your code and check the contents of variables while the code is running, a process that enables us to hunt for difficult bugs. You'll also take a briefer look at the debugging tools available for Firefox.

D'oh! I Can't Believe I Just Did That: Some Common Mistakes

There are seven common mistakes made by programmers. Some of these you'll learn to avoid as you become more experienced, but others may haunt you forever!

Chapter 4: Common Mistakes, Debugging, and Error Handling

You'll find it very useful in this chapter if your browser is set up to show errors. You did this in Chapter 2 in the section "Setting Up Your Browser for Errors." So if you don't already have error display set up, now would be a good time to do so.

Undefined Variables

JavaScript is actually very easygoing when it comes to defining your variables before assigning values to them. For example, the following will implicitly create the new global variable `abc` and assign it to the value 23:

```
abc = 23;
```

Although strictly speaking, you should define the variable explicitly with the `var` keyword like this:

```
var abc = 23;
```

Whether or not you use the `var` keyword to declare a variable has a consequence of what scope the variable has; so it is always best to use the `var` keyword. If a variable is used before it has been defined, an error will arise. For example, the following code will cause the error shown in Figure 4-1 in IE8 if the variable `abc` has not been previously defined (explicitly or implicitly):

```
alert(abc);
```

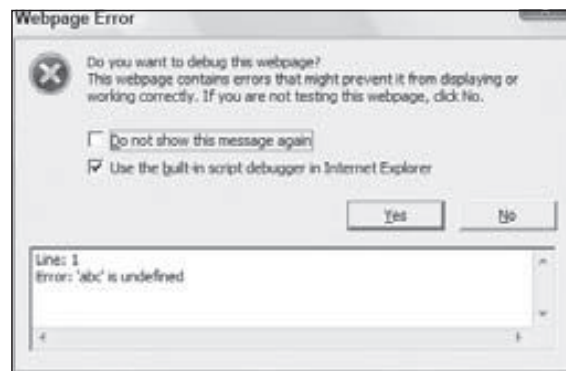


Figure 4-1

In Firefox you'll need to look in the JavaScript console, which you can view by choosing Tools ⇨ Error Console.

In addition, you must remember that function definitions also have parameters, which if not declared correctly can lead to the same type of error.

Take a look at the following code:

```
function foo(parametrOne)
{
    alert(parameterOne);
}
```

Chapter 4: Common Mistakes, Debugging, and Error Handling

If you call this function, you get an error message similar to the one shown in Figure 4-2.

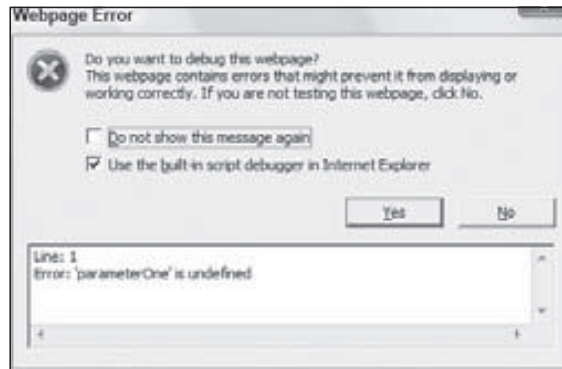


Figure 4-2

The error here is actually a simple typo in the function definition. The first parameter has the typo: it should read `parameterOne`, not `parametrOne`. What can be confusing with this type of error is that although the browser tells us the error is on one line, the source of the error is on another line.

Case Sensitivity

This is a major source of errors, particularly because it can be difficult to spot at times.

For example, spot the three case errors in the following code:

```
var myName = "Jeremy";
If (myName == "jeremy")
    alert(myName.toUpperCase());
```

The first error is the `if` keyword; the code above has `If` rather than `if`. However, JavaScript won't tell us that the error is an incorrect use of case, but instead IE will tell us `Object expected` and Firefox will tell us that `If` is not defined. Although error messages give us some idea of what's gone wrong, they often do so in an oblique way. In this case IE thinks you are trying to use an object called an `If` object and Firefox thinks you are trying to use an undefined function called `If`.

Okay, with that error cleared up, you come to the next error, not one of JavaScript syntax, but a logic error. Remember that `Jeremy` does not equal `jeremy` in JavaScript, so `myName == "jeremy"` is false, even though it's quite likely that you didn't care whether the word is `jeremy` or `jeremy`. This type of error will result in no error message at all, just the code not executing as you'd planned.

The third fault is with the `toUpperCase()` method of the `String` object contained in `myName`. The previous code uses `toUpperCase`, with the `C` in lowercase. IE will give us the message `Object doesn't support this property or method` and Firefox will report that `myName.toUpperCase` is not a function. On first glance it would be easy to miss such a small mistake and start checking your JavaScript reference guide for that method. You might wonder why it's there, but your code is not working. Again, you always need to be aware of case, something that even experts get wrong from time to time.

Incorrect Number of Closing Braces

In the following code, you define a function and then call it. However, there's a deliberate mistake. See if you can spot where it is.

```
function myFunction()
{
  x = 1;
  y = 2;
  if (x <= y)
  {
    if (x == y)
    {
      alert("x equals y");
    }
  }
  myFunction();
}
```

This is why formatting your code is important — you'll have a much easier time spotting errors such as this:

```
function myFunction()
{
  x = 1;
  y = 2;
  if (x <= y)
  {
    if (x == y)
    {
      alert("x equals y");
    }
  }
  myFunction();
}
```

Now you can see that the ending curly brace of the function is missing. When there are a lot of `if`, `for`, or `do while` statements, it's easy to have too many or too few closing braces. This type of problem is much easier to spot with formatted code.

Incorrect Number of Closing Parentheses

Take a look at the following code:

```
if (myVariable + 12) / myOtherVariable < myString.length)
```

Spot the mistake? The problem is the missing parenthesis at the beginning of the condition. You want `myVariable + 12` to be calculated before the division by `myOtherVariable` is calculated, so quite rightly you know you need to put it in parentheses.

```
(myVariable + 12) / myOtherVariable
```

Chapter 4: Common Mistakes, Debugging, and Error Handling

However, the `if` statement's condition must also be in parentheses. Not only is the initial parenthesis missing, but there is one more closing parenthesis than opening parentheses. Like curly braces, each opening parenthesis must have a closing parenthesis. The following code is correct:

```
if ((myVariable + 12) / myOtherVariable < myString.length)
```

It's very easy to miss a parenthesis or have one too many when you have many opening and closing parentheses.

Using Equals (=) Rather than Is Equal To (==)

Consider the following code:

```
var myNumber = 99;
if (myNumber = 101)
{
    alert("myNumber is 101");
}
else
{
    alert("myNumber is " + myNumber);
}
```

You'd expect, at first glance, that the `alert()` method in the `else` part of the `if` statement would execute, telling us that the number in `myNumber` is 99, but it won't. This code makes the classic mistake of using the assignment operator (`=`) instead of the equality operator (`==`). Hence, instead of comparing `myNumber` with 101, this code sets `myNumber` to equal 101. If you program in Visual Basic, or languages like it that use only one equals sign for both comparison and assignment, you'll find that every so often this mistake crops up. It's just so easy to make.

What makes things even trickier is that no error message is raised; it is just your data and logic that will suffer. Assigning a variable a value in an `if` statement may be perverse, but it's perfectly legal, so there will be no complaints from JavaScript. When embedded in a large chunk of code, a mistake like this is easily overlooked. Just remember it's worth checking for this error the next time your program's logic seems crazy.

Using a Method as a Property and Vice Versa

Another common error is where either you forget to put parentheses after a method with no parameters, or you use a property and do put parentheses after it.

When calling a method, you must always have parentheses following its name; otherwise, JavaScript thinks that it must be a pointer to the method or a property. For example, examine the following code:

```
var nowDate = new Date();
alert(nowDate.getMonth);
```

The first line creates an instance of the `Date` reference type. The second line attempts to call the `getMonth()` method of the newly created `Date` object, except the parentheses are missing. The following is the corrected code:

```
var nowDate = new Date();
alert(nowDate.getMonth());
```

Chapter 4: Common Mistakes, Debugging, and Error Handling

Just as you should always have parentheses after a method, you should never have parentheses after a property; otherwise, JavaScript thinks you are trying to use a method of that object:

```
var myString = "Hello, World!";
alert(myString.length());
```

The second line adds parentheses after the `length` property, making JavaScript think it is a method. This code should have been written like the following code:

```
var myString = new String("Hello");
alert(myString.length);
```

To compound the issue, it's common for a function to be passed as a parameter to another function (or a property as you'll see in Chapter 6 when working with events). In these situations, you pass the function without the opening and closing parentheses `()` at the end of the function name. Take a look at the following code:

```
function foo()
{
    alert("I'm in foo()!");
}

function bar(fpToCall)
{
    alert("Calling passed function").
    fpToCall();
}

bar(foo);
```

This code defines two functions: `foo()` and `bar()`. The `foo()` function simply displays a message box telling the user the `foo()` function is currently executing. The second function, `bar()`, accepts one argument that is a function. It displays a message saying it's calling the passed function, and then it executes that function. The final line calls the `bar()` function and passes a *pointer* of the `foo()` function. A pointer is a reference to a location in memory (we'll discuss memory references in the next chapter).

As a rule of thumb, use parentheses at the end of the function name when you want to execute the function, and leave the parentheses off when passing the function to another function or property.

Missing Plus Signs During Concatenation

In the following code, there's a deliberate concatenation mistake:

```
var myName = "Jeremy";
var myString = "Hello";
var myOtherString = "World";
myString = myName + " said " + myString + " " myOtherString;
alert(myString);
```

There should be a `+` operator between `" "` and `myOtherString` in the fourth line of code.

Although easy to spot in just a few lines, this kind of mistake can be harder to spot in large chunks of code. Also, the error message this type of mistake causes can be misleading. Load this code into a browser and you'll be told `Error : Expected by IE and Missing ; before statement` by Firefox. It's surprising how often this error crops up.

These most common mistakes are errors caused by the programmer. There are other types of errors, called *run-time errors*, that occur when your code executes in the browser, and they aren't necessarily caused by a typo, missing curly brace, parenthesis, or other pitfalls discussed. These types of errors can still be planned for, as you'll see in the next section.

Error Handling

When writing your programs, you want to be informed of every error. However, the last thing you want the user to see are error messages when you finally deploy the code to a web server for the whole world to access. Of course, writing bug-free code would be a good start, but keep the following points in mind:

- ❑ Occasions arise when conditions beyond your control lead to errors. A good example of this is when you are relying on something, such as a Java applet, that isn't on the user's computer and that you have no way of checking for.
- ❑ Murphy's Law states that anything that can go wrong will go wrong!

Preventing Errors

The best way to handle errors is to stop them from occurring in the first place. That seems like stating the obvious, but there are a number of things you should do if you want error-free pages.

- ❑ Thoroughly check pages in as many browsers as possible. This is easier said than done on some operating systems. The alternative is for you to decide which browsers you want to support for your web page, and then verify that your code works in them. Use the browser checking code found earlier in the book to send unsupported users to a nice, safe, and probably boring web page with reduced functionality, or maybe just supply them with a message that their browser and/or platform is not supported.
- ❑ Validate your data. If users can enter dud data that will cause your program to fail, then they will. Make sure that a textbox has data entered into it if your code fails if the text box is empty. If you need a whole number, you must make sure the user entered one. Is the date the user just entered valid? Is the e-mail address `mind your own business` the user just entered likely to be valid? No, so you must check that it is in the format `something@something.something`.

Okay, so let's say you carefully checked your pages and there is not a syntax or logic error in sight. You added data validation that confirms that everything the user enters is in a valid format. Things can still go wrong, and problems may arise that you can do nothing about. Here's a real-world example of something that can still go wrong.

One of your authors, Paul, created an online message board that relies on a small Java applet to enable the transfer of data to and from the server without reloading the page. Paul checked the code and everything was fine, and it continued to work fine after launching the board, except that in about five percent

Chapter 4: Common Mistakes, Debugging, and Error Handling

of cases the Java applet initialized but then caused an error due to the user being behind a particular type of firewall (a firewall is a means of stopping hackers from getting into a local computer network). There is no way of determining whether a user is behind a certain type of firewall, so there is nothing that can be done in that sort of exceptional circumstance. Or is there?

In fact, JavaScript includes something called the `try...catch` statement. This enables you to try to run your code; if it fails, the error is caught by the `catch` clause and can be dealt with as you wish. For the message board, Paul used a `try...catch` clause to catch the Java applet's failure and redirected the user to a more basic page that still displayed messages, but without using the applet.

The *try...catch* Statements

The `try...catch` statements work as a pair; you can't have one without the other. You use the `try` statement to define a block of code that you want to try to execute, and use the `catch` statement to define a block of code that will execute if an exception to the normal running of the code occurs in the block of code defined by the `try` statement. The term *exception* is key here; it means a circumstance that is extraordinary and unpredictable. Compare that with an *error*, which is something in the code that has been written incorrectly. If no exception occurs, the code inside the `catch` statement is never executed. The `catch` statement also enables you to get the contents of the exception message that would have been shown to the user had you not caught it first.

Let's create a simple example of a `try...catch` clause.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Try/Catch</title>
</head>
<body>
<script type="text/javascript">
try
{
    alert('This is code inside the try clause');
    alert('No Errors so catch code will not execute');
}
catch(exception)
{
    alert("The error is " + exception.message);
}
</script>
</body>
</html>
```

Save this as `trycatch.htm`.

This code first defines the `try` statement; as with all other blocks of code, you mark out the `try` block by enclosing it in curly braces.

Chapter 4: Common Mistakes, Debugging, and Error Handling

Next comes the `catch` statement. The code included `exception` in parentheses right after the `catch` statement. This `exception` is simply a variable name. It will store an object, of type `Error`, containing information about any exception thrown during code execution inside the `try` code block. We'll call this object the *exception object*. Although the word `exception` is used here, you can use any valid variable name. For example, `catch(exceptionObject)` would be fine and certainly more descriptive.

The exception object contains several properties that provide information about the exception that occurred. The bad news is the exception object in IE differs somewhat from the exception object in other browsers (and even Firefox, Opera, Safari, and Chrome have differing properties from each other!). The good news is there are similarities, and you don't have to worry about writing cross-browser code if you're only concerned with the exception's message and the type of exception.

All major browsers support the `name` and `message` properties. The `name` property contains the name of the error type, and the `message` property contains the error message the user would normally see. These properties are part of the ECMAScript 3 standard.

Back to the code at hand, within the curly braces after the `catch` statement is the code block that will execute if and only if an exception occurs. In this case, the code within the `try` code block is fine, and so the `alert()` method inside the `catch` block won't execute.

Insert a deliberate error.

```
try
{
    alert('This is code inside the try clause');
    ablert ('Exception will be thrown by this code');
}
catch(exception)
{
    alert("The error is " + exception.message);
}
```

Resave the document and reload the page in your browser. The first `alert()` method in the `try` block of code executes fine and the alert box will be displayed to the user. However, the second `ablert()` statement will cause an error and code execution will start at the first statement in the `catch` block.

If you're using Internet Explorer, the error description displayed will be `Object expected`. If you're using another browser, the same error is interpreted differently and reported as `ablert is not defined`.

If you change the code again, so that it has a different error, you'll see something important.

```
try
{
    alert('This is code inside the try clause');
    alert('This code won't work');
}
catch(exception)
{
    alert("The error is " + exception.message)
}
```

Chapter 4: Common Mistakes, Debugging, and Error Handling

Loading this revised code in a browser results in a normal browser error message telling you `Expected ') '` instead of displaying the alert box in the `catch` block. This happens because this code contains a syntax error; the functions and methods are valid, but you have an invalid character. The single quote in the word `won't` has ended the string parameter being passed to the `alert()` method. At that point JavaScript's syntax rules specify that a closing parenthesis should appear, which is not the case in this code. Before executing any code, the browser's JavaScript engine goes through all the code and checks for syntax errors, or code that breaches JavaScript's rules. If the engine finds a syntax error, the browser deals with it as usual; your `try` clause never runs and therefore cannot handle syntax errors.

Throwing Errors

The `throw` statement can be used within a `try` block of code to create your own run-time errors. Why create a statement to generate errors, when a bit of bad coding will do the same?

Throwing errors can be very useful for indicating problems such as invalid user input. Rather than using lots of `if...else` statements, you can check the validity of user input, then use `throw` to stop code execution in its tracks and cause the error-catching code in the `catch` block of code to take over. In the `catch` clause, you can determine whether the error is based on user input, in which case you can notify the user what went wrong and how to correct it. Alternatively, if it's an unexpected error, you can handle it more gracefully than with lots of JavaScript errors.

To use `throw`, type `throw` and include the error message after it.

```
throw "This is my error message";
```

Remember that when you catch the exception object in the `catch` statement, you can get hold of the error message that you have thrown. Although there's a string in this example `throw` statement, you can actually throw any type of data, including numbers and objects.

Try It Out try...catch and Throwing Errors

In this example you'll be creating a simple factorial calculator. The important parts of this example are the `try...catch` clause and the `throw` statements. It's a frameset page to enable you to demonstrate that things can go wrong that you can't do anything about. In this case, the page relies on a function defined within a frameset page, so if the page is loaded on its own, a problem will occur.

First let's create the page that will define the frameset and that also contains an important function.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Example</title>
<script type="text/javascript">
function calcFactorial(factorialNumber)
{
    var factorialResult = 1;
    for (; factorialNumber > 0; factorialNumber--)
    {
        factorialResult = factorialResult * factorialNumber;
    }
}
```

Chapter 4: Common Mistakes, Debugging, and Error Handling

```
        return factorialResult;
    }
</script>
</head>
<frameset cols="100%,*">
    <frame name="fraCalcFactorial" src="calcfactorial.htm" />
</frameset>
</html>
```

Save this page as `calcfactorialtopframe.htm`.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Example</title>
<script type="text/javascript">
function butCalculate_onclick()
{
    try
    {
        if (window.top.calcFactorial == null)
            throw "This page is not loaded within the correct frameset";
        if (document.form1.txtNum1.value == "")
            throw "!Please enter a value before you calculate its factorial";
        if (isNaN(document.form1.txtNum1.value))
            throw "!Please enter a valid number";
        if (document.form1.txtNum1.value < 0)
            throw "!Please enter a positive number";

        document.form1.txtResult.value =
            window.parent.calcFactorial(document.form1.txtNum1.value);
    }
    catch(exception)
    {
        if (typeof(exception) == "string")
        {
            if (exception.charAt(0) == "!")
            {
                alert(exception.substr(1));
                document.form1.txtNum1.focus();
                document.form1.txtNum1.select();
            }
            else
            {
                alert(exception);
            }
        }
        else
        {
            alert("The following error occurred " + exception.message);
        }
    }
}
}
```

Chapter 4: Common Mistakes, Debugging, and Error Handling

```
</script>
</head>
<body>
<form action="" name="form1">
  <input type="text" name="txtNum1" size="3" /> factorial is
  <input type="text" name="txtResult" size="25" /><br />
  <input type="button" value="Calculate Factorial"
    name="butCalculate" onclick="butCalculate_onclick()" />
</form>
</body>
</html>
```

Save this page as `calcfactorial.htm`. Then load the first page, `calcfactorialtopframe.htm`, into your browser.

The page consists of a simple form with two text boxes and a button. Enter the number 4 into the first box and click the Calculate Factorial button. The factorial of 4, which is 24, will be calculated and put in the second text box (see Figure 4-3.)

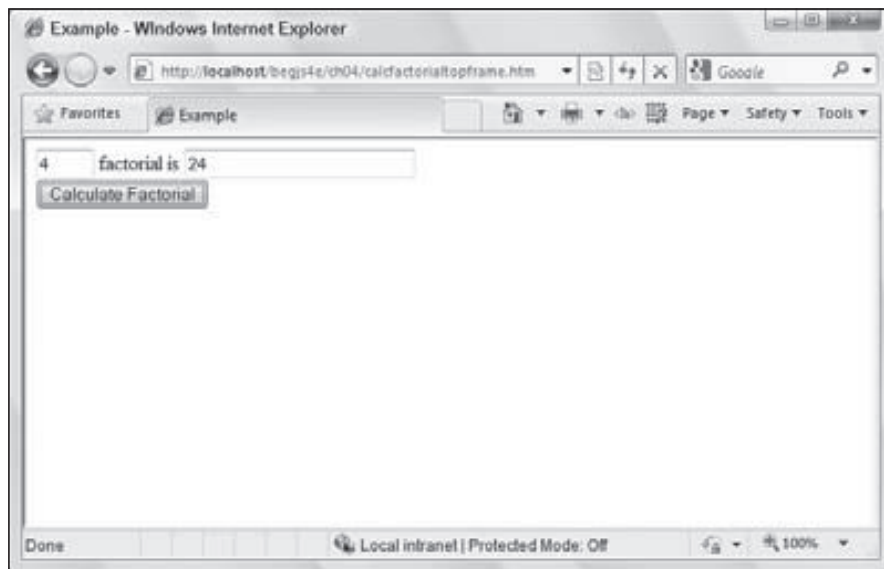


Figure 4-3

The factorial of a number is the product of all the positive integers less than or equal to that number. For example, the factorial of 4 (written $4!$) is $1 * 2 * 3 * 4 = 24$. Factorials are used in various branches of mathematics, including statistics. Here, you want only to create a function that does something complex enough to be worthy of a function, but not so complex as to distract you from the main purpose of this example: the `try...catch` and `throw` statements.

If you clear the first text box and click the Calculate Factorial button, you'll be told that a value needs to be entered. If you enter an invalid non-numeric value into the first text box, you'll be told to enter a valid value. If you enter a negative value, you'll be told to enter a positive value.

Also, if you try loading the page `calcfactorial.htm` into your browser and enter a value in the text box and click the Calculate Factorial button, you'll be told that the page is not loaded into the correct frameset.

Chapter 4: Common Mistakes, Debugging, and Error Handling

As you'll see, all of these error messages are created using the `try...catch` and `throw` statements.

Because this example is all about `try...catch` and `throw`, you'll concentrate just on the `calcfactorial.htm` page, in particular the `butCalculate_onclick()` function, which is connected to the `onclick` event handler of the form's only button.

Start by looking at the `try` clause and the code inside it. The code consists of four `if` statements and another line of code that puts the calculated factorial into the second text box. Each of the `if` statements checks for a condition that, if true, would cause problems for your code.

The first `if` statement checks that the `calcFactorial()` function, in the top frameset window, actually exists. If not, it throws an error, which is caught by the `catch` block. If the user loads the `calcfactorial.htm` page rather than the frameset page `calcfactorialtopframe.htm`, then without this `throw` statement your code will fail.

```
try
{
    if (window.top.calcFactorial == null)
        throw "This page is not loaded within the correct frameset";
```

The next three `if` statements check the validity of the data entered into the text box by the user. First make sure the user entered something into the text box; then make sure the user entered a number, and then finally check that the value is not negative. Again if any of the `if` conditions is true, you throw an error, will be caught by the `catch` block. Each of the error messages you define starts with an exclamation mark, the purpose of which is to mark the error as a user input error, rather than an error such as not being in a frameset.

```
    if (document.form1.txtNum1.value == "")
        throw "Please enter a value before you calculate its factorial";
    if (isNaN(document.form1.txtNum1.value))
        throw "Please enter a valid number";
    if (document.form1.txtNum1.value < 0)
        throw "Please enter a positive number";
```

If everything is fine, the `calcFactorial()` function will be executed and the results text box will be filled with the factorial of the number entered by the user.

```
        document.form1.txtResult.value =
            window.parent.calcFactorial(document.form1.txtNum1.value);
    }
```

Finally, turn your attention to the `catch` part of the `try...catch` statement. First, any message thrown by the `try` code will be caught by the exception variable.

```
catch(exception)
{
```

The type of data contained in `exception` will depend on how the error was thrown. If it was thrown by the browser and not by your code, `exception` will be an object, the exception object. If it's thrown by your code, then in this instance you've thrown only primitive strings. So the first thing you need to do is decide what type of data `exception` contains. If it's a string, you know it was thrown by your code and can deal with it accordingly. If it's an object, and given that you know none of your code throws

Chapter 4: Common Mistakes, Debugging, and Error Handling

objects, you assume it must be the browser that has generated this exception and that `exception` is an `Exception` object.

```
if (typeof(exception) == "string")
{
```

If it was code that generated the exception using a `throw` (and so `exception` is a string), you now need to determine whether the error is a user input error, such as the text box not containing a value to calculate, or whether it was another type of error, such as the page not being loaded in your frameset. All the user input exception messages had an exclamation mark at the beginning, so you use an `if` statement to check the first character. If it is a `!`, you notify the user of the error and then return focus to your control. If it's not, you just display an error message.

```
    if (exception.charAt(0) == "!")
    {
        alert(exception.substr(1));
        document.form1.txtNum1.focus();
        document.form1.txtNum1.select();
    }
    else
    {
        alert(exception);
    }
}
```

If `exception` was not a string, you know you have an exception object and need to display the message property:

```
    else
    {
        alert("The following error occurred " + exception.message);
    }
}
```

Nested try...catch Statements

So far you've been using just one `try...catch` statement, but it's possible to include a `try...catch` statement inside another `try` statement. Indeed, you can go further and have a `try...catch` inside the `try` statement of this inner `try...catch`, or even another inside that, the limit being what it's actually sensible to do.

So why would you use nested `try...catch` statements? Well, you can deal with certain errors inside the inner `try...catch` statement. If, however, you're dealing with a more serious error, the inner catch clause could pass that error to the outer catch clause by throwing the error to it.

Here's an example:

```
try
{
    try
    {
        ablurt("This code has an error");
    }
}
```

Chapter 4: Common Mistakes, Debugging, and Error Handling

```
}
catch(exception)
{
    var eName = exception.name;

    if (eName == "TypeError" || eName == "ReferenceError")
    {
        alert("Inner try...catch can deal with this error");
    }
    else
    {
        throw exception;
    }
}
}
catch(exception)
{
    alert("Error the inner try...catch could not handle occurred");
}
```

In this code you have two `try...catch` pairs, one nested inside the other.

The inner `try` statement contains a line of code that contains an error. The `catch` statement of the inner `try...catch` checks the value of the error's name. If the exception's name is either `TypeError` or `ReferenceError`, the inner `try...catch` deals with it by way of an alert box (see Appendix B for a full list of error types and their descriptions). Unfortunately, and unsurprisingly, the type of error thrown by the browser depends on the browser itself. In the preceding example, IE reports the error as a `TypeError` whereas the other browsers report it as a `ReferenceError`.

If the error caught by the inner `catch` statement is any other type of error, it is thrown up in the air again for the `catch` statement of the outer `try...catch` to deal with.

Let's change the `butCalculate_onclick()` function from the previous example, `calcfactorial.htm`, so that it has both an inner and an outer `try...catch`.

```
function butCalculate_onclick()
{
    try
    {
        try
        {
            if (window.top.calcFactorial == null)
                throw ("This page is not loaded within the correct frameset");
            if (document.form1.txtNum1.value == "")
                throw ("!Please enter a value before you calculate its factorial");
            if (isNaN(document.form1.txtNum1.value))
                throw ("!Please enter a valid number");
            if (document.form1.txtNum1.value < 0)
                throw ("!Please enter a positive number");
            document.form1.txtResult.value =
                window.parent.calcFactorial(document.form1.txtNum1.value);
        }
    }
}
```

Chapter 4: Common Mistakes, Debugging, and Error Handling

```
        catch(exception)
        {
            if (typeof(exception) == "string" && exception.charAt(0) == "!")
            {
                alert(exception.substr(1));
                document.form1.txtNum1.focus();
                document.form1.txtNum1.select();
            }
            else
            {
                throw exception;
            }
        }
    }
}
catch(exception)
{
    switch (exception)
    {
        case "This page is not loaded within the correct frameset":
            alert(exception);
            break;
        default :
            alert("The following critical error has occurred \n" + exception);
    }
}
}
```

The inner `try...catch` deals with user input errors. However, if the error is not a user input error thrown by us, it is thrown for the outer `catch` statement to deal with. The outer `catch` statement has a `switch` statement that checks the value of the error message thrown. If it's the error message thrown by us because the `calcfactorialtopframe.htm` is not loaded, the `switch` statement deals with it in the first case statement. Any other error is dealt with in the `default` statement. However, there may well be occasions when there are lots of different errors you want to deal with in `case` statements.

***finally* Clauses**

The `try...catch` statement has a `finally` clause that defines a block of code that will execute whether or not an exception was thrown. The `finally` clause can't appear on its own; it must be after a `try` block, which the following code demonstrates:

```
try
{
    ablurt("An exception will occur");
}
catch(exception)
{
    alert("Exception occurred");
}
finally
{
    alert("Whatever happens this line will execute");
}
```


The `finally` part is a good place to put any cleanup code that needs to be executed regardless of any errors that occurred previously.

You've seen the top mistakes made by developers, and you've also seen how to handle errors in your code. Unfortunately, errors will still occur in your code, so let's take a look at one way to make remedying them easier by using a debugger.

Debugging

JavaScript is traditionally looked upon as a difficult language to write and debug due to the lack of decent development tools. This is not the case now, however, thanks to many tools made available to developers. Most notably are the debugging tools available for Internet Explorer, Firefox, Safari, and Opera. With these tools, you can halt the execution of your script with breakpoints and then step through code line by line to see exactly what is happening.

You can also find out what data is being held in variables and execute statements on the fly. Without debuggers, the best you can do is use the `alert()` method in your code to show the state of variables at various points.

Debugging is generally universal across all browsers, and even languages. Some debugging tools may offer more features than others, but for the most part, the following concepts can be applied to any debugger:

- ❑ Breakpoints tell the debugger it should break, or pause code execution, at a certain point. You can set a breakpoint anywhere in your JavaScript code, and the debugger will halt code execution when it reaches the breakpoint.
- ❑ Watches allow you to specify variables that you want to inspect when your code pauses at a breakpoint.
- ❑ The call stack is a record of what functions and methods have been executed to the breakpoint.
- ❑ The console allows you to execute JavaScript commands in the context of the page and within the scope of the breakpoint. In addition, it catalogs all JavaScript errors found in the page.
- ❑ Stepping is the most common procedure in debugging. It allows you to execute one line of code at a time. There are three ways to step through code.
 - ❑ Step Into executes the next line of code. If that line is a function call, the debugger executes the function and halts at the first line of the function.
 - ❑ Step Over, like Step Into, executes the next line of code. If that line is a function, Step Over executes the entire function and halts at the first line outside the function.
 - ❑ Step Out returns to the calling function when you are inside a called function. Step Out resumes the execution of code until the function returns. It then breaks at the return point of the function.

Before delving into the various debuggers, let's create a page you can debug. Note the deliberate typo in line 16. Be sure to include this typo if creating the page from scratch.

Chapter 4: Common Mistakes, Debugging, and Error Handling

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Debug: Times Table</title>
    <script type="text/javascript">
        function writeTimesTable(timesTable)
        {
            var counter;
            var writeString;
            for (counter = 1; counter < 12; counter++)
            {
                writeString = counter + " * " + timesTable + " = ";
                writeString = writeString + (timesTable * counter);
                writeString = writeString + "<br />";
                documents.write(writeString);
            }
        }
    </script>

</head>
<body>
    <script type="text/javascript">
        writeTimesTable(2);
    </script>
</body>
</html>
```

Save this as `debug_timestable.htm`.

The next section walks you through the features and functionality of the Firebug add-on for Firefox. Because of the universal nature of debugging and debuggers, the sections for Internet Explorer, Safari, and Opera will merely familiarize you with the UI for each browser's debugger and point out any differences.

Debugging in Firefox with Firebug

For years, the only JavaScript debugger for Firefox was a Mozilla project codenamed Venkman. Its feature-set resembled that of Microsoft's Script Editor, but many developers felt Venkman wasn't user-friendly. One such developer, Joe Hewitt, decided to write his own debugger using the built-in debugging API (application programming interface) in Firefox. He christened his creation Firebug, and the rest, as they say, is history. Today, Firebug is the defacto JavaScript debugger (and much more!) for Firefox, and all other JavaScript (and web development) tools for other browsers are based, in principle, on Firebug.

Unfortunately, Firebug does not come with Firefox by default. Instead, you have to install the Firebug addon. You can download the latest version of Firebug from <http://www.getfirebug.com>, from Joe Hewitt's website at <http://www.johewitt.com/software/firebug/>, or from Mozilla's add-on site at <https://addons.mozilla.org/en-US/firefox/addon/1843>.

To install Firebug, open Firefox and go to either of the provided URLs. Click the Install button on the web page, and follow the instructions. Be sure to restart Firefox after Firebug's installation.

Chapter 4: Common Mistakes, Debugging, and Error Handling

You can access Firebug a couple of ways. You can click the Firebug icon in the status bar in the lower-right corner of the Firefox window. If you do not have Firefox's status bar visible, you can open Firebug by selecting Firebug ⇨ Open Firebug from the Tools menu in Firefox. By default, Firebug opens as a panel in Firefox (see Figure 4-4).

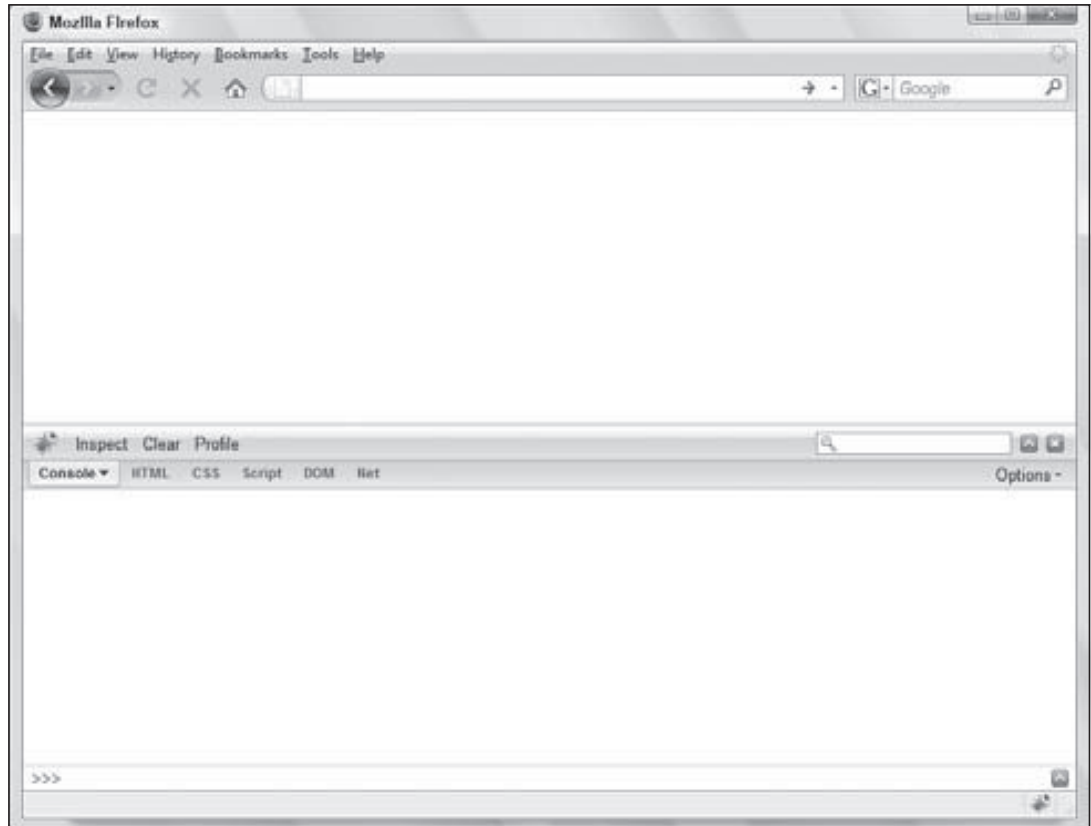


Figure 4-4

You can pop it out to its own window by clicking the up arrow next to the Close button.

Open `debug_timestable.htm` in Firefox. If the status bar is visible, you should see red text in the lower-right corner of the Firefox window stating “1 Error.” Click that message (or go through the Tools menu to open Firebug), and Firebug will open to the console. The console serves multiple purposes in Firebug; it lists JavaScript errors, and it also allows you to execute JavaScript code on the fly. We’ll play with the console later.

The JavaScript debugger is contained in the Script tab, and it is made up of two panels. The left panel contains the source code, and the right panel contains three different views to choose from: Breakpoints, Watch, and Stack.

- ☐ **Breakpoints:** Lists all breakpoints that you’ve created for the code in the current page.
- ☐ **Watch:** Lists the variables in scope and their values at the breakpoint. You can also add other variables to watch.
- ☐ **Stack:** Displays the call stack.

Chapter 4: Common Mistakes, Debugging, and Error Handling

The source code in the left panel is read-only; if you want to change it, you have to edit the file in your text editor. Let's do so and change the offending `documents` in line 16 to `document`. Save it, and reload the web page.

Having corrected the mistake and reloaded the page, you should see the times table in your web page, as shown in Figure 4-5.

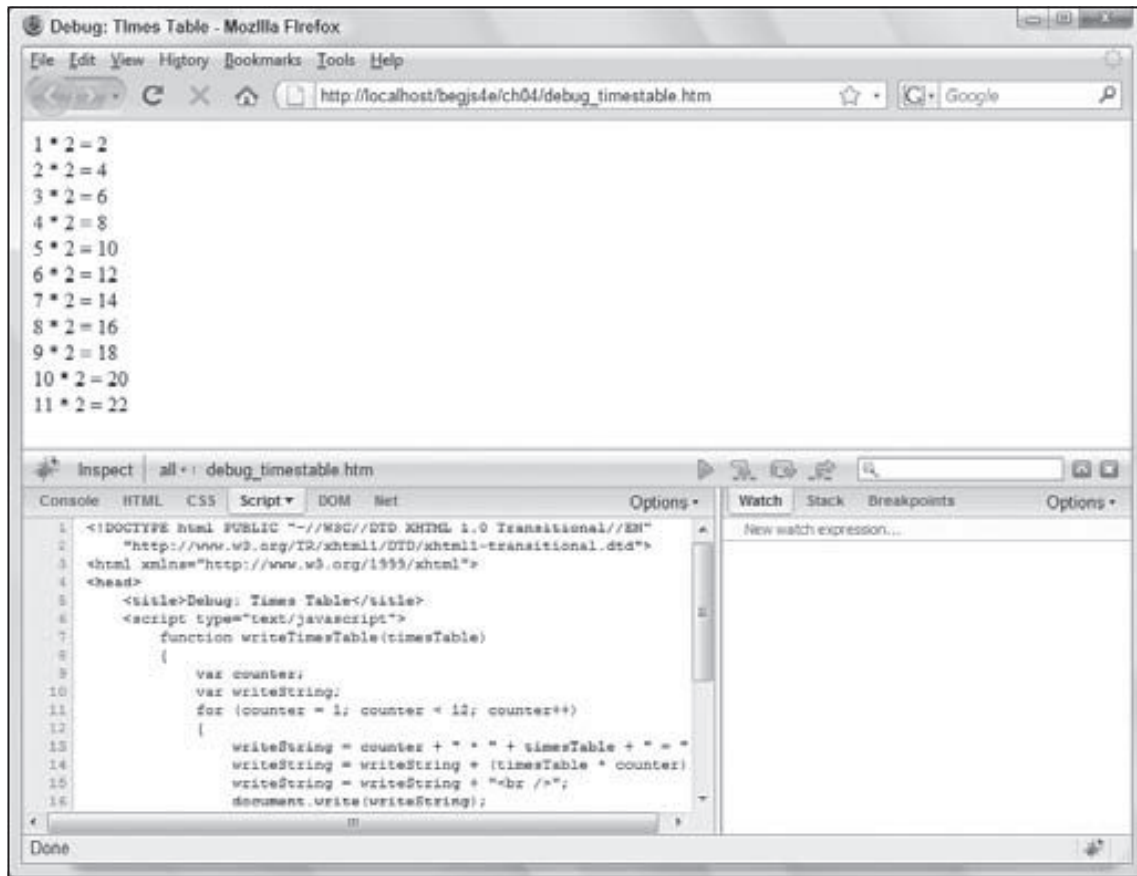


Figure 4-5

Also notice that the source code in Firebug's left panel updated to reflect your changes.

Setting Breakpoints

As mentioned earlier, breakpoints tell the debugger to pause code execution at a specific point in your code. This is handy when you want to inspect your code while it executes. Creating breakpoints in Firebug is straightforward; simply left-click in the gray area to the left of the source code's line numbers (the gutter). Breakpoints are denoted by a red circle in the gutter where you clicked.

You can also create a breakpoint when writing your code by using the `debugger` keyword (we'll use this a bit later).

Chapter 4: Common Mistakes, Debugging, and Error Handling

Keeping the corrected `debug_timestable.htm` loaded in Firefox, create a breakpoint on line 14.

```
writeString = writeString + (timesTable * counter);
```

Reload the page, and notice Firebug stopped code execution at the breakpoint you just created. Firebug highlights the current line of code in light yellow and puts a yellow arrow in the gutter. This line hasn't been executed yet.

Click the Breakpoints tab in the right panel; it shows you the list of breakpoints (only one in this case). Each entry in the list consists of a checkbox to enable/disable the breakpoint, the containing function's name, the file name and line number of the source file, the source text of the breakpoint, and a Delete button.

Now click the Watch tab.

Watches

The Watch tab displays variables and their values currently in scope at the current line while code execution is paused. Figure 4-6 shows the contents of the Watch tab at this breakpoint.

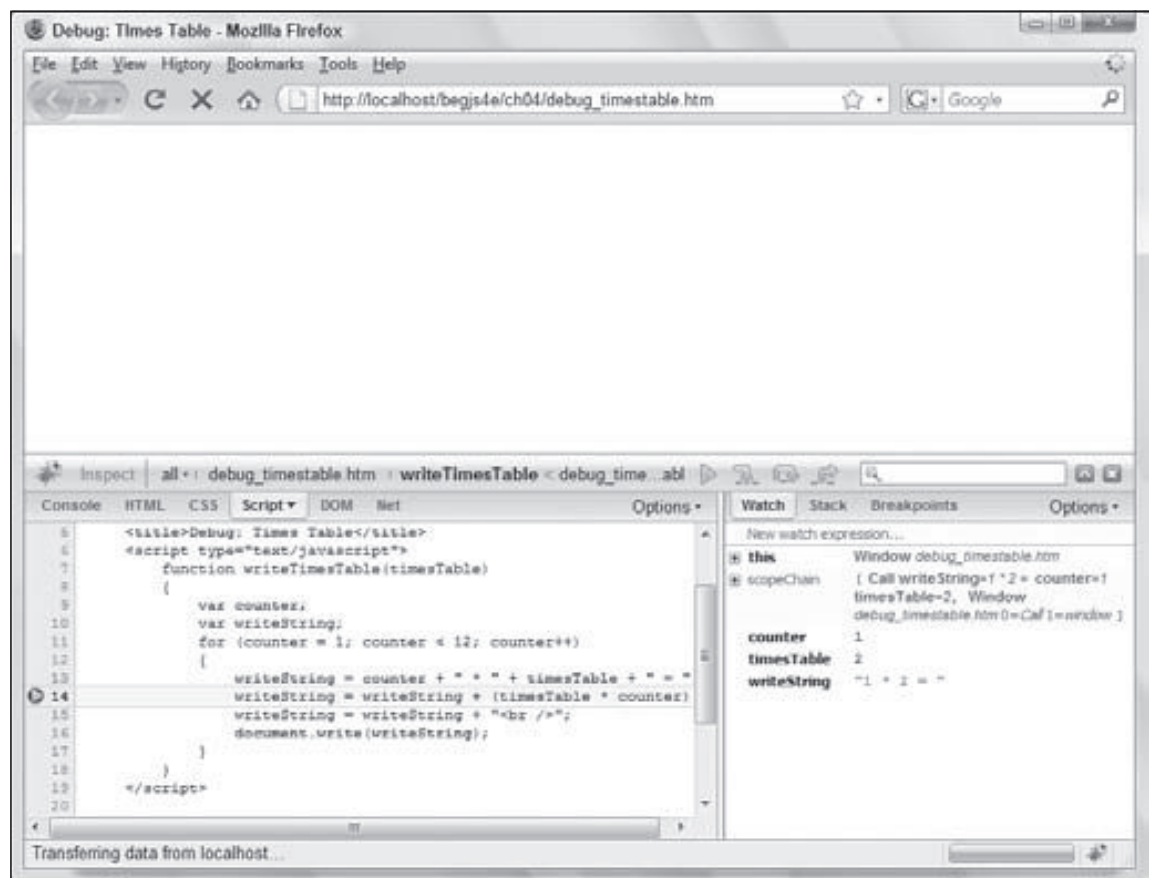


Figure 4-6

Notice that the `counter`, `timesTable`, and `writeString` variables are visible (as is `this`).

Chapter 4: Common Mistakes, Debugging, and Error Handling

You can also add your own variables to watch, inspecting their values as you step through code. To add a watch, simply click “New watch expression...” type the variable name you want to watch, and press the Enter key. Watches that you add have a gray background, and moving your mouse over them reveals a red Delete button.

You can watch any variable you want. If the variable is in scope, the variable’s value is displayed. If the variable is out of scope, a `ReferenceError` is displayed as its value.

Although this information is helpful when you want to see what exactly is going on in your code, it’s not very helpful if you can’t control code execution. It’s impractical to set a breakpoint and reload the page multiple times just to advance to the next line, so we use a process called stepping.

Stepping Through Code

Code stepping is controlled by four buttons in the upper-right of the window, next to the source code search box (see Figure 4-7).

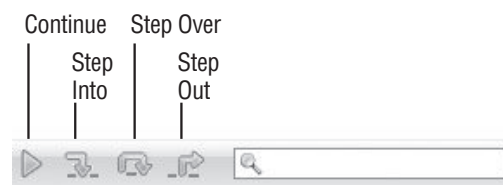


Figure 4-7

- ❑ **Continue** (shortcut key is F8): Its function is to continue code execution until either the next breakpoint or the end of all code is reached.
- ❑ **Step Into** (shortcut key is F11): Executes the current line of code and moves to the next statement. If the current line is a function, then it steps to the first line of the function.
- ❑ **Step Over** (F10): Like Step Into, this executes the current line of code and moves to the next statement. However, if the statement is a function, it executes the function and steps to the next line after the function call.
- ❑ **Step Out**: Returns to the calling function.

Let’s do some stepping; follow these steps:

1. Step Into the code by clicking the icon or pressing F11. The debugger executes the currently highlighted line of code and moves to the next line.
2. Look in the Watch tab and at the value of `writeString`; it is “1 * 2 = 2”. As you can see, the values displayed in the Watch tab are updated in real time.
3. One nice feature of Firebug is the page updates, if necessary, as you step through code. Click Step Into two more times to see this in action. Figure 4-8 shows the page updated while stepping through code.

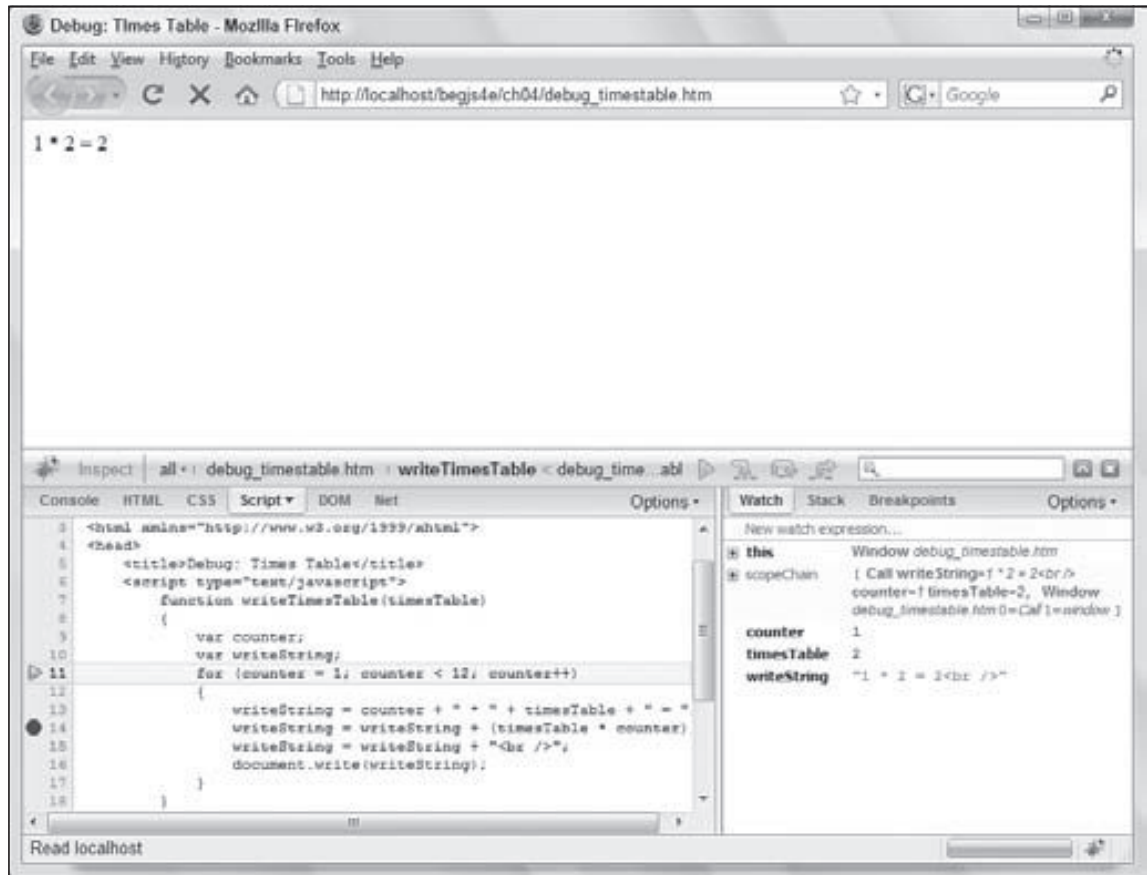


Figure 4-8

You may find that the function you stepped into is not the source of the bug and want to execute the remaining lines of code in the function to continue step by step from the point at which the function was called. Do so by clicking the Step Out icon to step out of the code. However, if you're in a loop and the breakpoint is set inside the loop, you will not step out of the function until you iterate through the loop.

There may also be times when you have some code with a bug in it that calls a number of functions. If you know that some of the functions are bug-free, then you may want to just execute those functions instead of stepping into them and seeing them executed line by line. Use Step Over in these situations to execute the code within a function but without going through it line by line.

Alter your times-table code in `debug_timestable.htm` as follows so you can use it for the three kinds of stepping:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>Av
  <title>Debug: Times Table 2</title>

  <script type="text/javascript">
```


Chapter 4: Common Mistakes, Debugging, and Error Handling

```
function writeTimesTable(timesTable)
{
    var counter;
    var writeString;
    for (counter = 1; counter < 12; counter++)
    {
        writeString = counter + " * " + timesTable + " = ";
        writeString = writeString + (timesTable * counter);
        writeString = writeString + "<br />";
        document.write(writeString);
    }
}
</script>

</head>
<body>
    <script type="text/javascript">
        var timesTable;
        for (timesTable = 1; timesTable <= 12; timesTable++)
        {
            document.write("<p>")
            writeTimesTable(timesTable)
            document.write("</p>")
        }
    </script>
</body>
</html>
```

Save this as `debug_timestable2.htm`. Note that there are no errors in this HTML file.

The following instructions will walk you through the process of stepping through code.

1. Set a breakpoint in line 26, the `for` loop in the body of the page, and reload the page.
2. Click the Step Into icon and code execution will move to the next statement. Now the first statement inside the `for` loop, `document.write("<p>")`, is up for execution.
3. When you click the Step Into icon again, it will take you to the first calling of the `writeTimesTable()` function.
4. You want to see what's happening inside that function, so click Step Into again and you'll step into the function. Your screen should look like the one shown in Figure 4-9.
5. Click the Step Into icon a few times to get the gist of the flow of execution of the function. In fact, stepping through code line by line can get a little tedious. So let's imagine you're happy with this function and want to run the rest of it.
6. Use Step Out to run the rest of the code. The function has been fully executed, and you're back the calling line, as you can see from Figure 4-10.
7. Click the Step Into icon twice to execute `document.write()` (it won't be visible because it's a closing tag).
8. Click Step Into four more times. Execution will continue through the condition and incrementing parts of the `for` loop, ending back at the line that calls the `writeTimesTable()` function.

Chapter 4: Common Mistakes, Debugging, and Error Handling

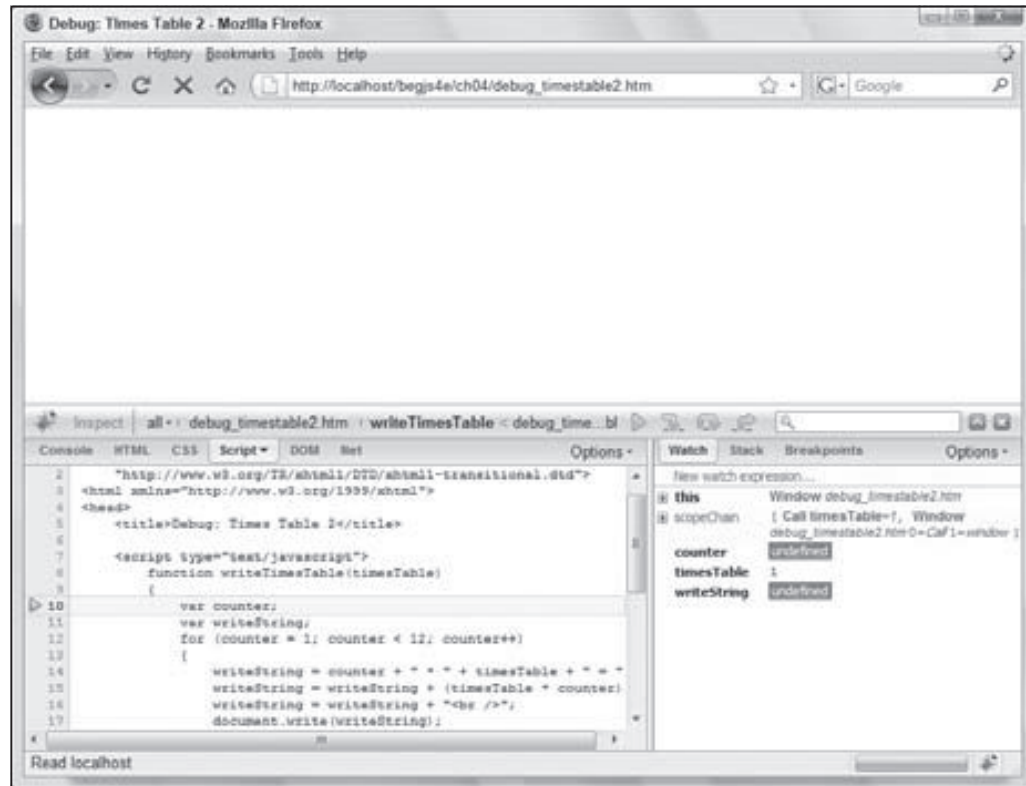


Figure 4-9

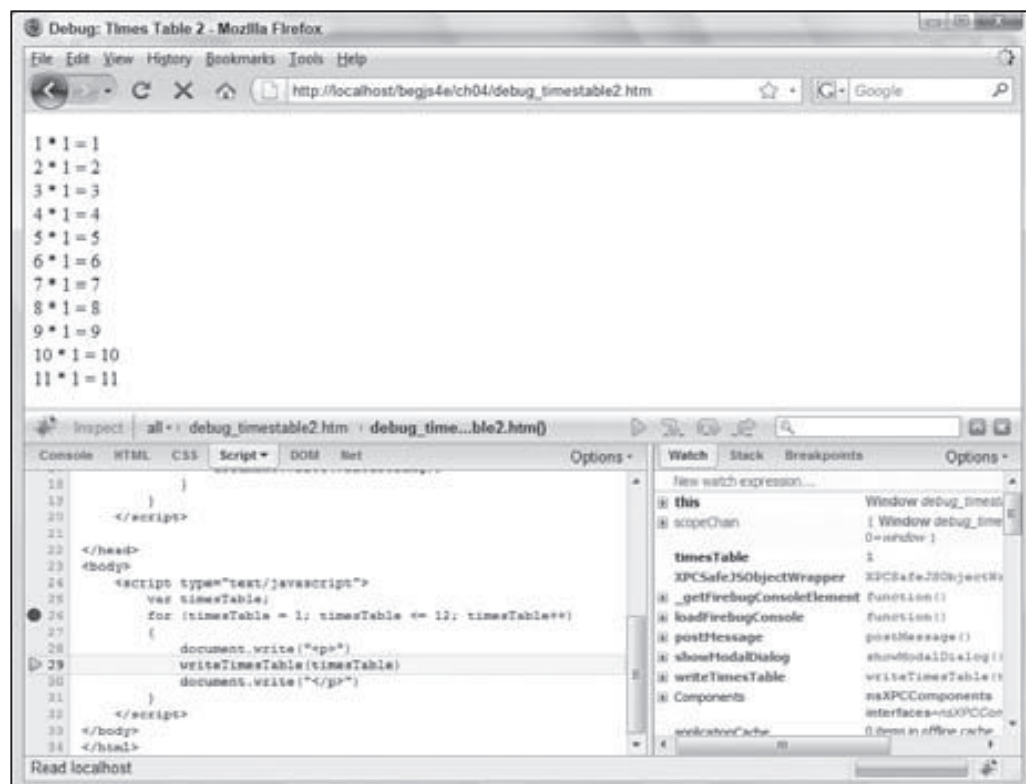


Figure 4-10

Chapter 4: Common Mistakes, Debugging, and Error Handling

9. You've already seen this in action, so really you want to step over it and go to the next line. Well, no prizes for guessing that Step Over is what you need to do. Click the Step Over icon (or press the F10 key) and the function will be executed, but without stepping through it statement by statement. You should find yourself back at the `document.write("</p>")` line.

If you've finished debugging, you can run the rest of the code without stepping through each line by clicking the Continue icon (or pressing F8) on the toolbar. You should see a page of times tables from $1 \times 1 = 1$ to $11 \times 12 = 132$ in the browser.

The Console

While you're stepping through code and checking its flow of execution, what would be really useful is the ability to evaluate conditions and even to change things on the fly. You can do these things using the console.

Follow these steps:

1. Remove the previously set breakpoint by clicking the red circle in the source code panel and set a new breakpoint at line 17:

```
document.write(writeString);
```
2. Let's see how you can find out the value currently contained in the variable `writeString`. Reload the page. When the debugger stops at the breakpoint, click the Console tab, click in the ">>>" field, and type the name of the variable you want to examine, in this case `writeString`. Press the Enter key. This will cause the value contained in the variable to be printed below your command in the command window, as shown in Figure 4-11.

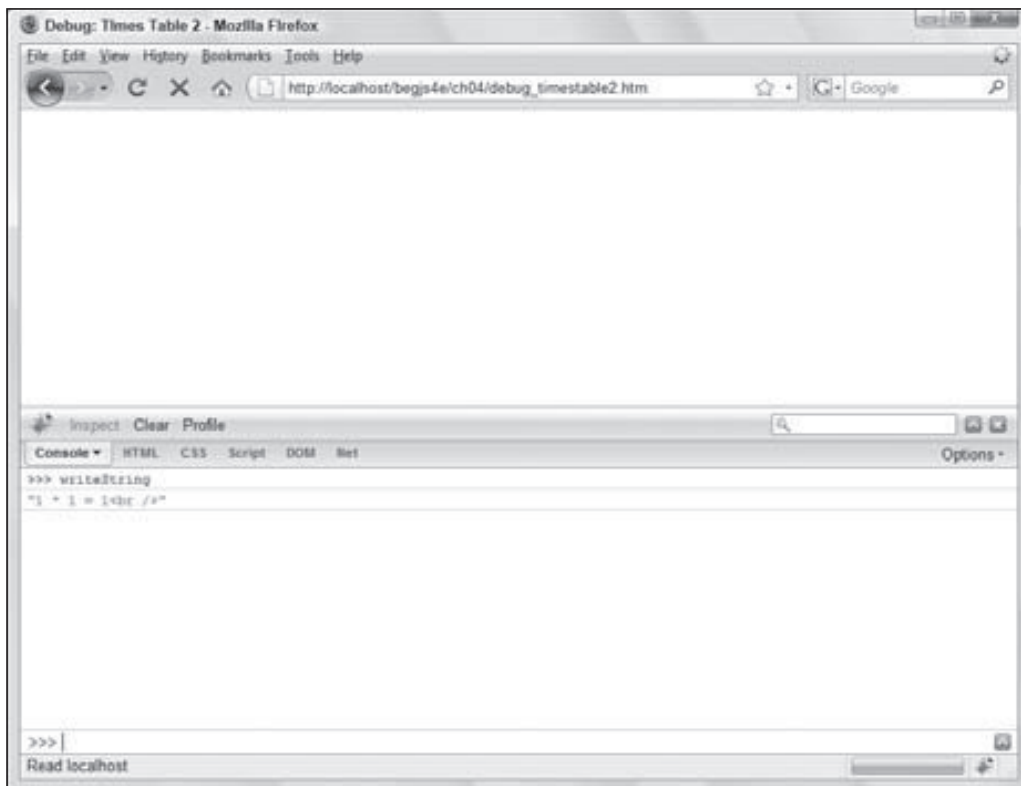


Figure 4-11

Chapter 4: Common Mistakes, Debugging, and Error Handling

3. If you want to change a variable, you can write a line of JavaScript into the command window and press Enter. Try it with the following code:

```
writeString = "Changed on the Fly<br />"
```

4. Click the Script tab, and remove the breakpoint by clicking the red circle and then clicking the Continue icon. You see the results of your actions: where the 1*1 times table result should be, the text you changed on the fly has been inserted.

This alteration does not change your actual HTML source file, just the page currently loaded in the browser.

The console can also evaluate conditions. Recreate the breakpoint on line 26 and reload the page. Leave execution stopped at the breakpoint, and Step Into the `for` loop's condition.

Type the following into the command window and press Enter:

```
timesTable <= 12
```

Because this is the first time the loop has been run, as shown in Figure 4-12, `timesTable` is equal to 1 so the condition `timesTable <= 12` evaluates to `true`.

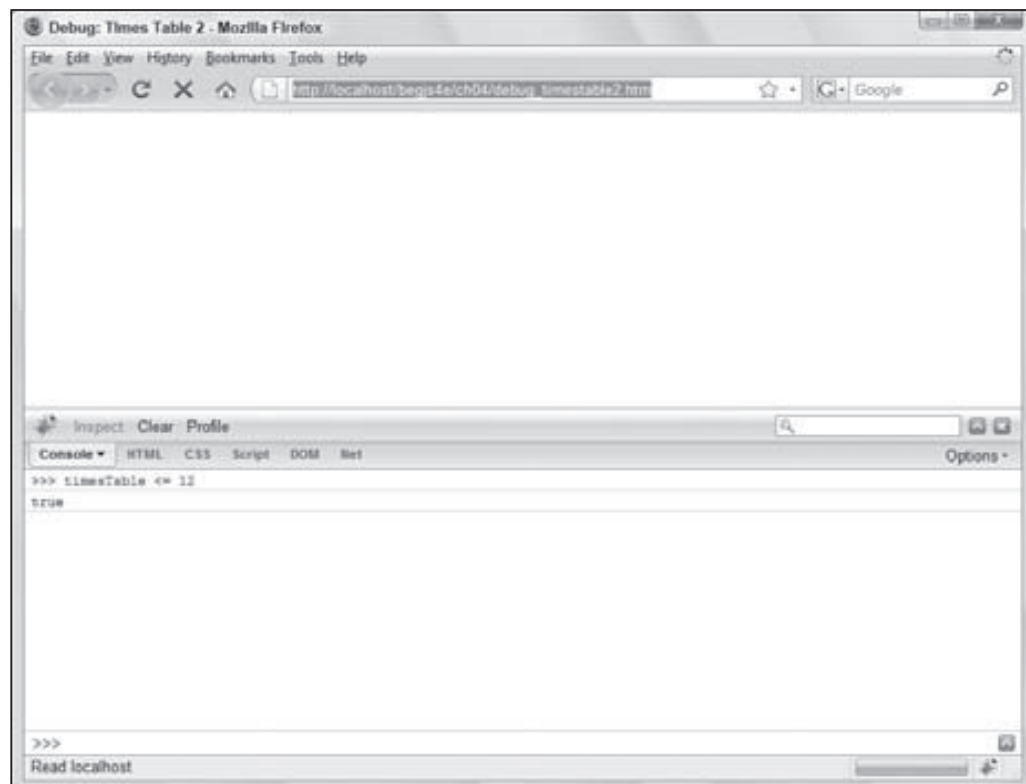


Figure 4-12

You can also use the console to access properties of the Browser Object Model (something we'll cover in Chapter 6). For example, if you type `window.location.href` into the command window and press Enter, it will tell you the web page's URL.

Chapter 4: Common Mistakes, Debugging, and Error Handling

The console isn't limited to single lines of JavaScript. Click the up arrow on the right side of the screen and you can enter multiple statements to execute.

Call Stack Window

When you are single-stepping through the code, the call stack window keeps a running list of which functions have been called to get to the current point of execution in the code.

Let's create an example web page that demonstrates the call stack very nicely.

1. Enter this code:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Debugging: Callstack</title>
    <script type="text/javascript">
        function firstCall()
        {
            secondCall();
        }

        function secondCall()
        {
            thirdCall();
        }

        function thirdCall()
        {
            //
        }

        function button1_onclick()
        {
            debugger
            firstCall();
        }
    </script>
</head>
<body>
    <input type="button" value="Button" name="button1"
        onclick="return button1_onclick()" />
</body>
</html>
```

- 2.** Save this page as `debug_callstack.htm`, and load it into Firefox. All you'll see is a blank web page with a button.
- 3.** Click the button and the debugger will open at the `debugger` statement in the `button1_onclick()` function, which is connected to the button's `onclick` event handler.
- 4.** Click the Call Stack tab in the right panel. Your debugger now looks like what is shown in Figure 4-13.

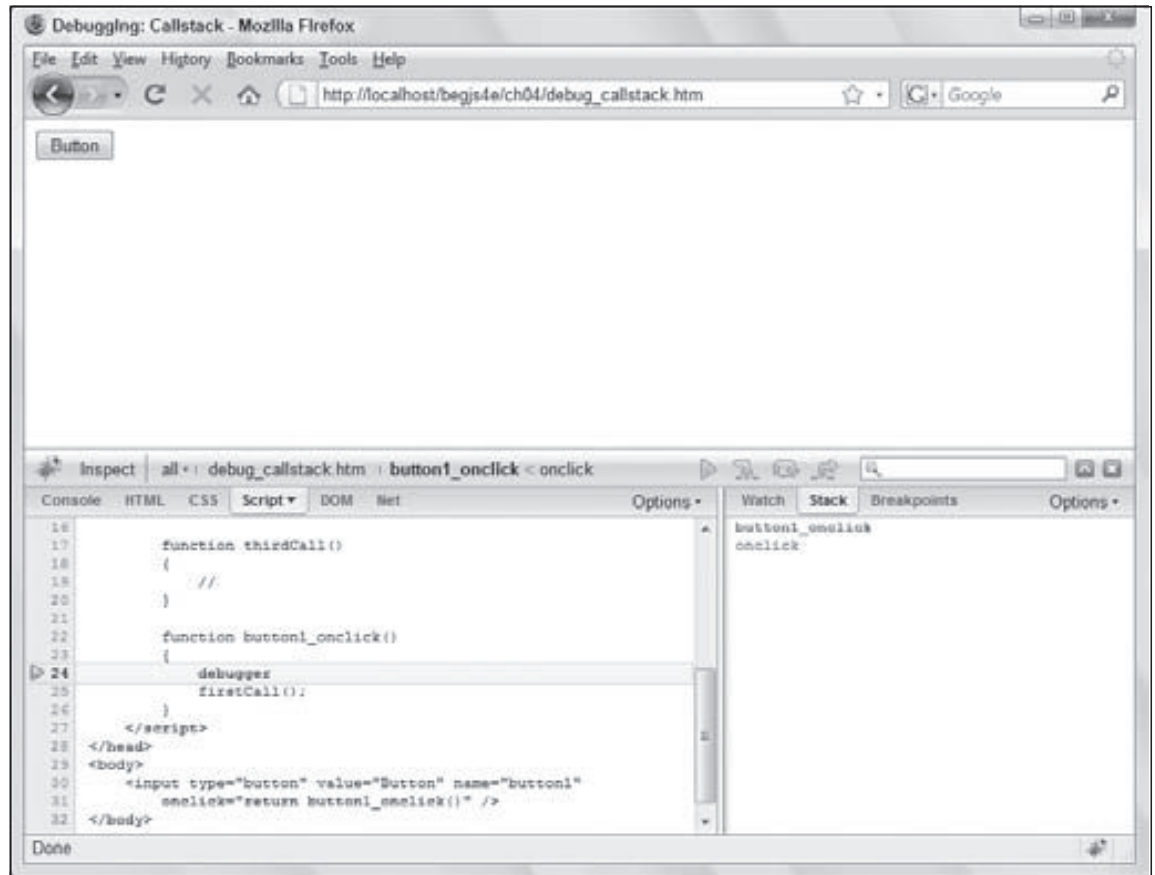


Figure 4-13

Firebug adds the function to the top of the call stack for every function call. You can already see that the first function called was actually the code attached to the `onclick` event handler of your button. Next, added to the call stack is the function called by the `onclick` event handler, which is the function `button1_onclick()` shown at the top of the call stack.

5. If you want to see where each function was first entered, just click the function name in the call stack window. Click `onclick` and the calling code (that is, the code connected to the `onclick` attribute of the `<input/>` element) will be shown. Now click the top line, `button1_onclick`, and that will take you back to the current execution point.
6. Now Step Into twice. The first step is to the line that calls the `firstCall()` function. The second step takes you into that function itself. The function is immediately added to the call stack, as shown in Figure 4-14.
7. Step Into again to enter the second function, `secondCall()`. Again this is added to the call stack. One more step takes you into the third function, `thirdCall()`, again with its name being added to the top of the call stack.

Chapter 4: Common Mistakes, Debugging, and Error Handling

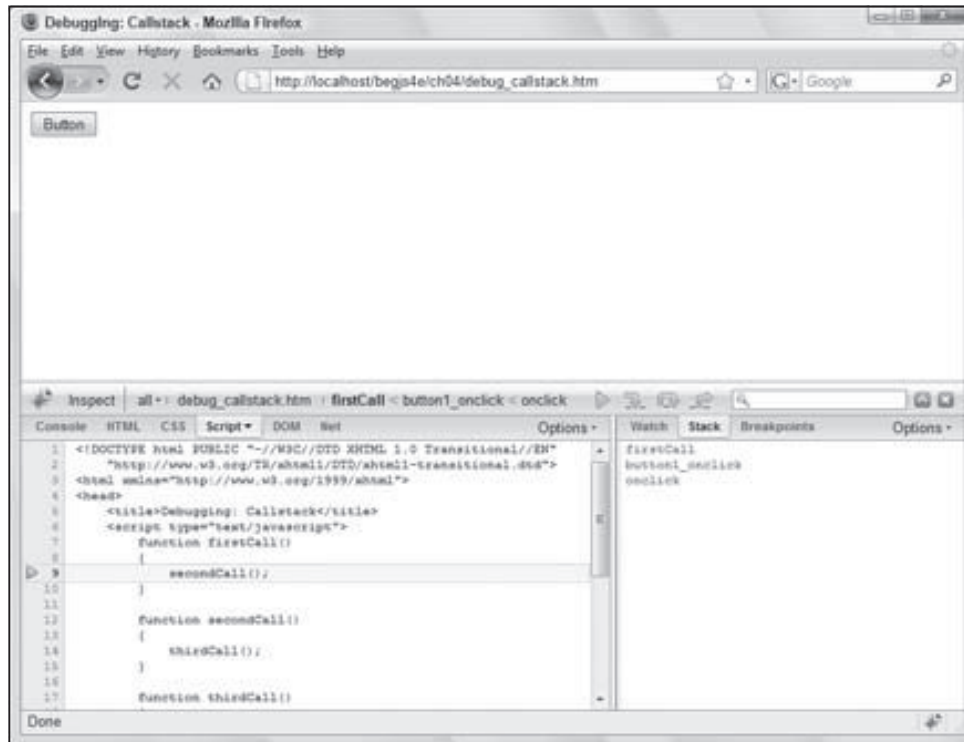


Figure 4-14

8. Step Into again, and as you leave the function `thirdCall()` you will see that its name is removed from the top of the call stack. Yet another step takes you out of the second function `secondCall()`, whose name is also now removed from the stack. Each additional click takes you out of a function, and removes its name from the call stack, until eventually all the code has been executed and you're back to the browser again.

This demo page was very simple to follow, but with complex pages, especially multi-frame pages, the call stack can prove very useful for tracking where you are, where you have been, and how you got there.

As mentioned earlier, most other developer tools for other browsers are based upon Firebug, and you'll soon see this with IE8's built-in tools.

Debugging in Internet Explorer

Before version 8, developers had to download and install the Microsoft Script Debugger for any type of script debugging. Thankfully, Microsoft built a debugger into IE8, but it is turned off by default. To enable it, follow these steps:

1. Click Tools ⇨ Internet Options.
2. Click the Advanced tab, and uncheck the box next to "Disable script debugging (Internet Explorer)" under the Browsing section (see Figure 4-15).
3. Click OK to save the settings and exit the Internet Options dialog box.

Chapter 4: Common Mistakes, Debugging, and Error Handling



Figure 4-15

You can access the debugger in a couple of ways now that it is enabled:

- ❑ You can bring up the debugger by clicking Tools ⇨ Developer Tools. The Developer Tools contains a variety of tools you might find useful (like Firebug, it's much more than a JavaScript debugger). For easy access, consider modifying the command bar to include the Developer Tools button. Once the Developer Tools window appears, click the Script tab as shown in Figure 4-16.

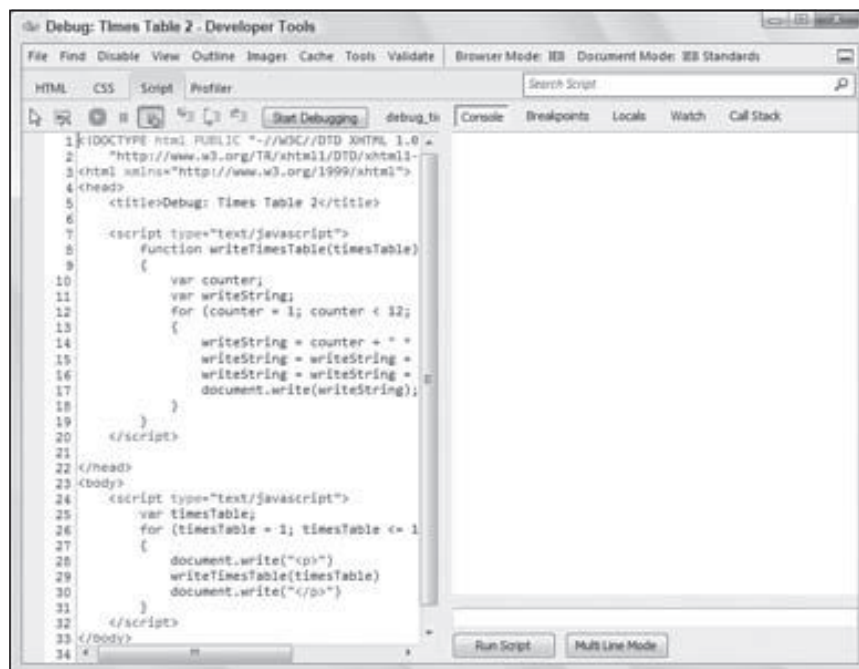


Figure 4-16

Chapter 4: Common Mistakes, Debugging, and Error Handling

- ❑ This method is probably the easiest way to begin debugging. Simply navigate to the desired page. If any errors occur, a dialog box appears asking if you want to debug (see Figure 4-17).

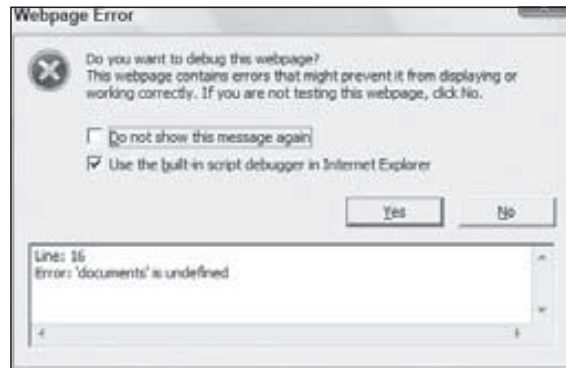


Figure 4-17

Click the Yes button, and the Developer Tools window appears with the Script tab already selected. The debugger stops on the line where the error is and highlights it in yellow, although this may not be obvious from the black-and-white screenshot in Figure 4-18. Go ahead and load the original version of `debug_timestable.htm` in IE8 (the one with the error) to see this in action. Click Yes to start debugging.

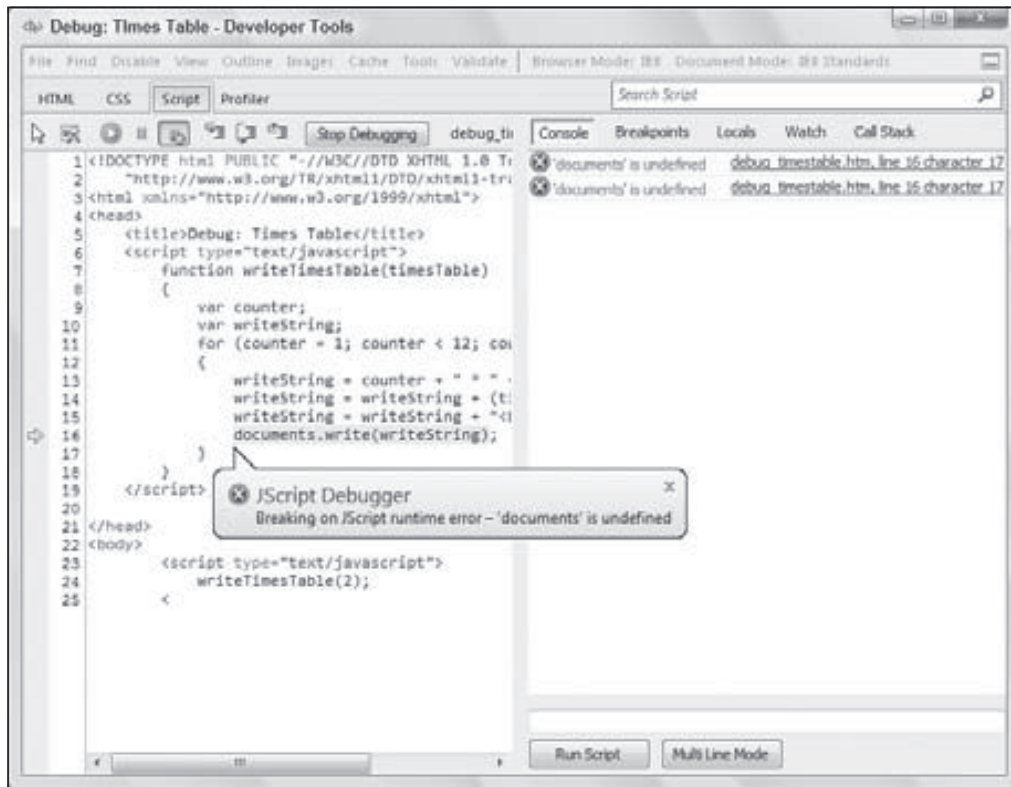


Figure 4-18

Chapter 4: Common Mistakes, Debugging, and Error Handling

One of the primary differences in IE8's debugger is it must be in debugging mode if you want to debug. This may seem obvious, but if you recall, Firebug didn't have a debugging mode.

As you can see in Figure 4-18, the debugger is made up of two panels. The left displays the source code of the file that contains the error. If multiple files contain JavaScript, you can select those files by using the pull-down menu next to the Stop Debugging button.

The right panel contains five tabs:

- ☐ **Console:** Consists of an upper and lower panel. The lower panel allows you to enter and run JavaScript code. It has a single-line mode and a multi-line mode for larger blocks of code. The upper panel is a log of commands you entered along with their results.
- ☐ **Breakpoints:** Lists all breakpoints that you've created for the code in the current page.
- ☐ **Locals:** Lists the variables and their values in scope of the breakpoint.
- ☐ **Watches:** Lists the variables and their values you specify to watch at the breakpoint.
- ☐ **Call Stack:** Displays the call stack.

Another difference in IE8's Developer Tools is the additional Locals tab. Microsoft took Firebug's Watch tab and broke it into the Locals and Watches tabs.

The source code in the left panel is read-only, so changing it requires editing the file in your text editor. Do so, and change the offending `documents` in line 16 to `document`. Save it and try to reload the web page. Notice you cannot do so. This is because the debugger is currently running and stopped at an error. In order to reload the page, you must click the Stop Debugging button. With the debugger now stopped, you can reload the page.

Having corrected the mistake and reloaded the page, you should see the times table in your web page.

Setting Breakpoints

Creating a breakpoint in IE8 is as simple and straightforward as it is in Firebug; simply click in the gutter on the line you want the debugger to break at. After creating a breakpoint, you'll notice a red circle next to the line number.

Upon creating a breakpoint, an entry is added in the list of breakpoints found by clicking the Breakpoints tab. Each entry consists of a checkbox to enable/disable the breakpoint, the file name of the source file, and the line number the breakpoint is on. Figure 4-19 shows a breakpoint on line 17 of `debug_timestable2.htm`.

Chapter 4: Common Mistakes, Debugging, and Error Handling

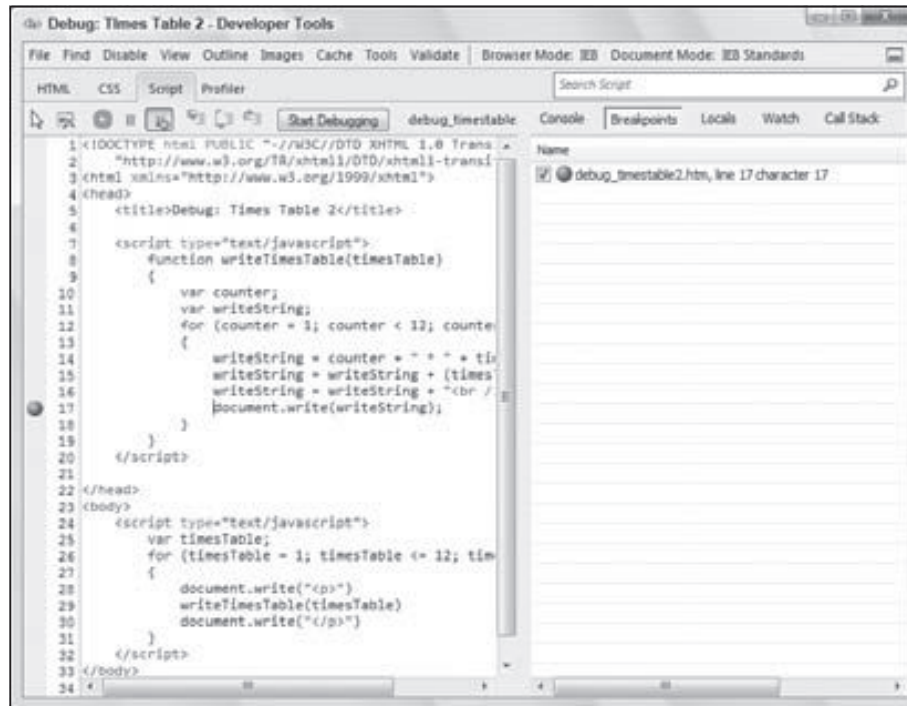


Figure 4-19

IE8's JavaScript debugger also supports the `debugger` keyword. However, you must be in debugging mode in order for IE8's debugger to pause on that line. Otherwise, you'll be greeted with the old selection screen. If you use the `debugger` keyword in your code and see Figure 4-20, then you need to turn on debugging mode.



Figure 4-20

Adding Watches

The Watch tab's sole purpose is to list the variables you want to watch and display their values and type. Adding a watch is slightly different in IE8 than it is in Firebug: You must be in debugging mode and stopped at a breakpoint to add a watch.

If you are in debugging mode and stopped at a breakpoint, simply click "Click to Add..." type the variable you want to watch, and press the Enter key (see Figure 4-21).

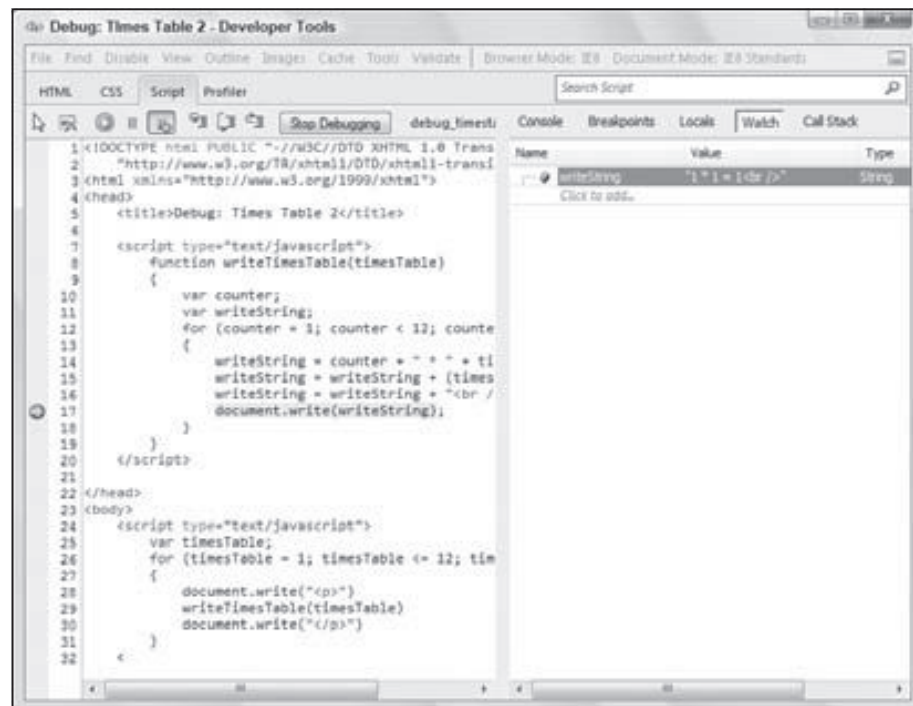


Figure 4-21

Stepping Through Code

At the top of the debugger window, and to the left of the Debugging button, are six buttons that control code execution (see Figure 4-22).

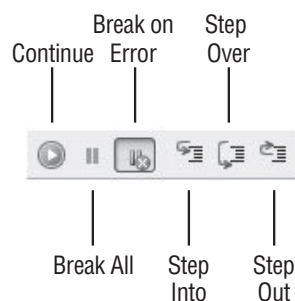


Figure 4-22

Chapter 4: Common Mistakes, Debugging, and Error Handling

The Continue option (shortcut key F5) continues code execution until either the next breakpoint or the end of all code. The second option, Break All, ensures that the debugger breaks before the execution of the next statement. Break on Error tells the debugger to break whenever an error occurs. Step Into (F11), Step Over (F10), and Step Out (Shift+F11) behave as they do in Firebug.

IE8's debugger denotes the current line by highlighting the line in yellow and adds a yellow arrow in the gutter.

Unlike Firefox and Firebug, stepping through code does not update the web page. The JavaScript executes, but you will not see the results until all code is executed.

The Console

Unlike Firebug's console, IE8's Developer Tools console is located with the rest of the JavaScript tools and is accessed via the Console tab, but that's where the primary differences end.

The console logs JavaScript errors and allows you to execute code within the context of the line the debugger is stopped at. Figure 4-23 shows the "Changed on the Fly" example from the Firebug section recreated in IE8.

By default, the console accepts only single lines of JavaScript code. You can change this by clicking the Multi Line Mode button.

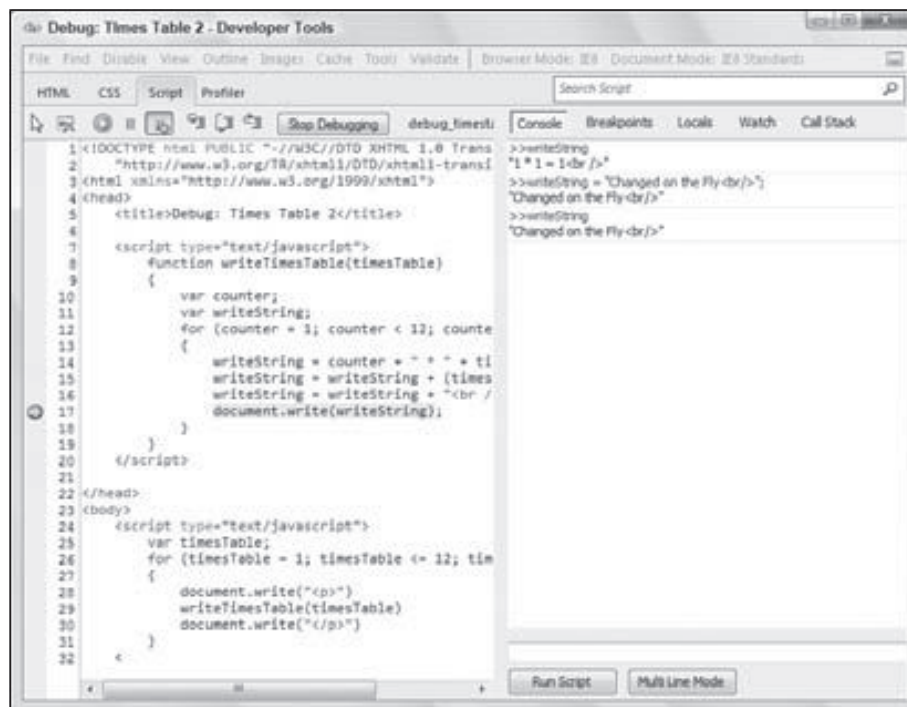


Figure 4-23

Debugging in Safari

Safari's debugging tool's story is similar to that of IE's. Safari's rendering engine is called Webkit, and the folks that write and maintain Webkit built a separate tool, codenamed Drosera, that contained the tools similar to Firebug and IE8's Developer Tools. It was a separate download, and it required you to attach it to a specific Safari/Webkit window.

Safari 3 includes a tool called Web Inspector, but it does not have any JavaScript debugging capability. Starting with Safari 4, the Web Inspector has a built-in JavaScript debugger, which we'll cover in this section.

Chrome also uses Webkit, but only version 3 beta includes the script debugger at the time of this writing.

Like IE8, the Web Inspector is disabled by default. To enable it, follow these steps:

1. Click the Settings menu button and choose the Preferences option (see Figure 4-24).

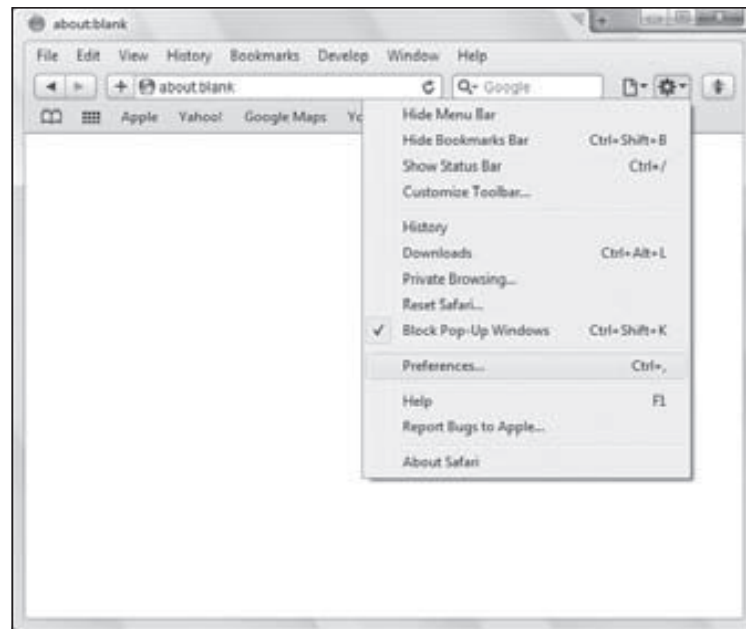


Figure 4-24

2. In the Preferences window, click the Advanced tab and select the Show Develop Menu in Menu Bar option (see Figure 4-25). Close the Preferences window.

Chapter 4: Common Mistakes, Debugging, and Error Handling

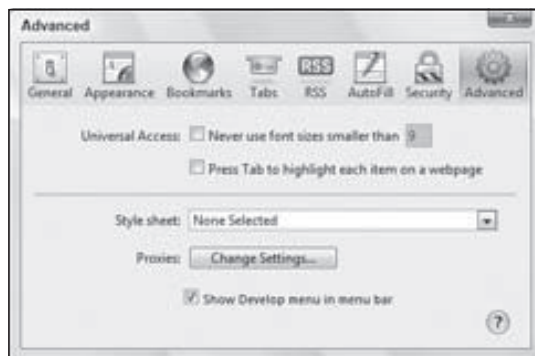


Figure 4-25

3. Click the Settings menu button and select the Show Menu Bar option. This will display the traditional menus at the top of the window.
4. To open the debugger, select Develop ⇨ Start Debugging JavaScript from the menu bar.

When the window opens, you'll see some code that definitely isn't yours. That's OK — you can change that in a bit. First, let's look at the window and identify the separate parts. Figure 4-26 shows the JavaScript debugger when it was first opened on the `debug_timestable2.htm` file. The code displayed may vary on your computer.

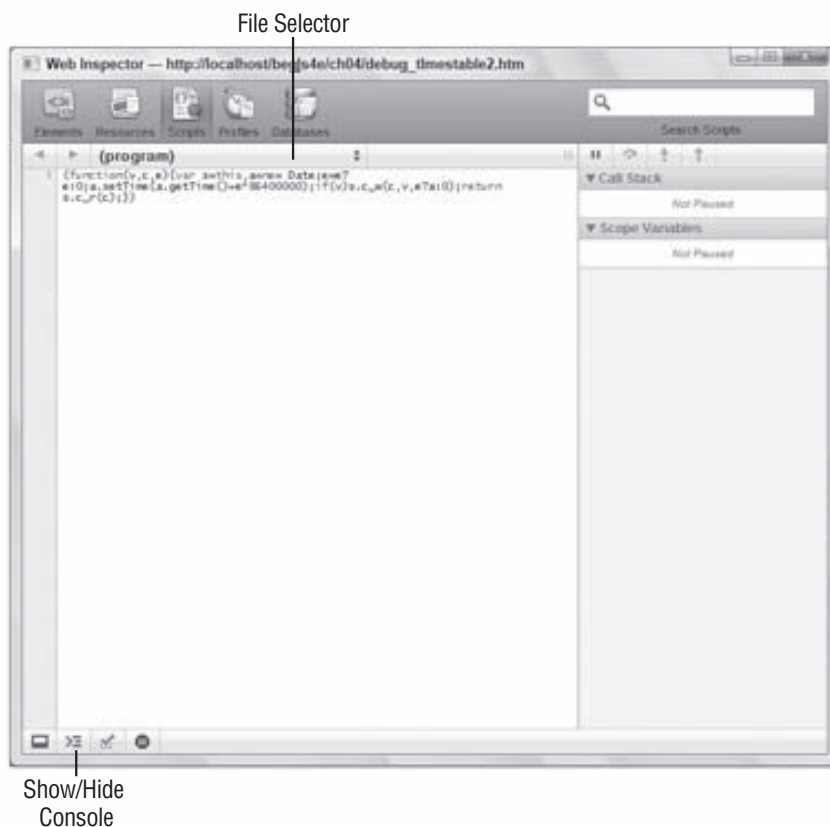


Figure 4-26

Chapter 4: Common Mistakes, Debugging, and Error Handling

Unlike the tools you looked at previously, Safari's Web Inspector doesn't use tabs to partition the various features. Instead, it organizes everything within the window so you have easy access to all features. You can see the Call Stack and Scope Variables are not tabs, but rather individual sections you can view at the same time as you debug. Click the Console button, and you'll see that it adds a panel to the bottom of the window (see Figure 4-27).

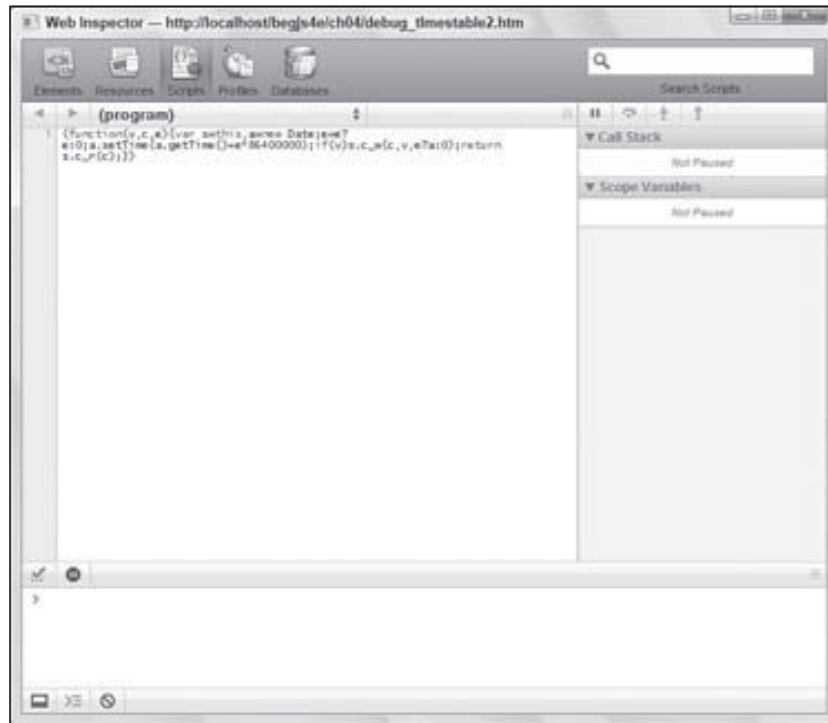


Figure 4-27

Everything is here, readily available and visible to you; so load `debug_timestable2.htm` in Safari and go back to the Web Inspector window. Click the file selection drop-down menu and choose the HTML file to display the source code of the file. Like the previous tools, the source code is read-only, but you can set breakpoints.

Setting Breakpoints

Creating a breakpoint follows the same procedure in Web Inspector as the other tools: click in the gutter on the line you want the debugger to break at. Breakpoints in Web Inspector are denoted by a blue tag (see Figure 4-28). Create one on line 17.

Unlike Firebug and IE's Developer Tools, Web Inspector does not list the breakpoints you set in a separate area, so remember where your breakpoints are if you use Safari and Web Inspector as your browser and debugger of choice.

Reload the page so the debugger can break and we can walk through the features.

Web Inspector supports the `debugger` keyword.

Chapter 4: Common Mistakes, Debugging, and Error Handling

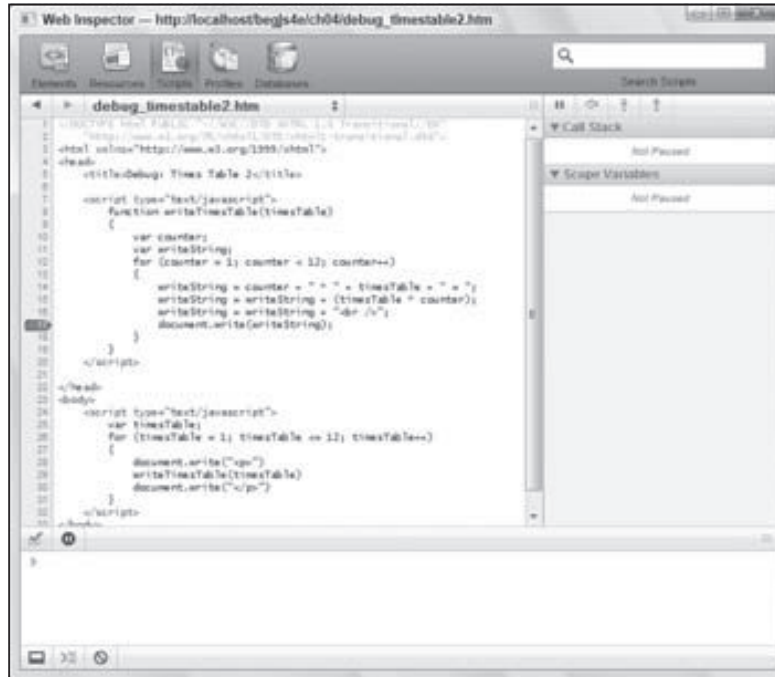


Figure 4-28

No Watches for You!

Web Inspector does not allow you to add your own variables to watch, but the Scope Variables section displays all variables in scope. Figure 4-29 shows how the variables are divided into local and global variables.

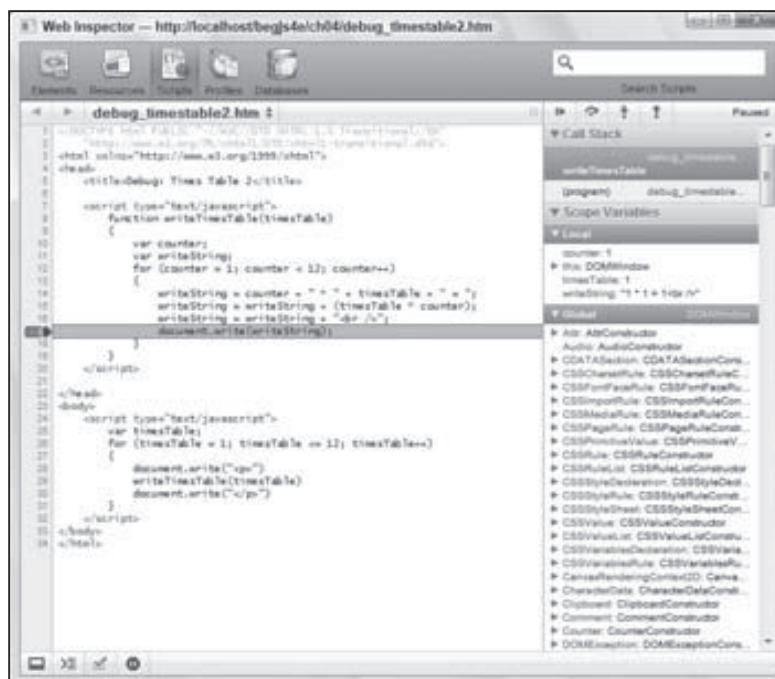


Figure 4-29

Stepping Through Code

The code-stepping buttons are at the top of the right panel and underneath the search box (see Figure 4-30).

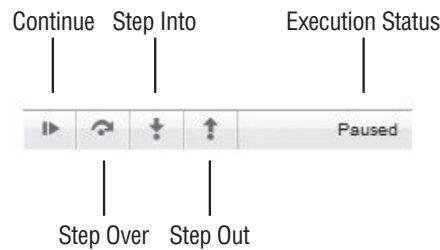


Figure 4-30

These buttons perform the same functions as in Firebug and IE; however, they are in a slightly different order. The first button continues code execution until either the next breakpoint or the end of all code. The second button is Step Over, the third is Step Into, and the fourth is Step Out.

Like Firefox and Firebug, Safari and Web Inspector update the page as you step through code. So you can see the results as each line executes.

The Console

The console serves the same purpose as it does in the previous tools. You can check the value of a variable by typing the variable and pressing the Enter key. You can also execute code in the context of the current line of code. Try the “Changed on the Fly” example from the Firebug section to see it in action.

Unlike the previous tools, the Web Inspector console does not allow for multi-line input.

Although the Web Inspector’s UI is sleek and tab-less (and some would say cluttered), it’s time to venture back into the world of tabs with Opera’s Dragonfly.

Using Dragonfly: Opera’s Development Tools

Opera’s Dragonfly is a latecomer to the realm of browser-based development tools. At the time of this writing, it is currently pre-beta software, but it comes included with Opera as of version 9.5.

There are two ways to open Dragonfly:

- ❑ Through the Tools menu: Tools ⇨ Advanced ⇨ Developer Tools.
- ❑ Through the Debug menu, which can be installed by opening the following URL in Opera:
`http://dragonfly.opera.com/app/debugmenu/DebugMenu.ini.`

Figure 4-31 shows Dragonfly open with `debug_timestable2.htm` loaded in Opera.

Chapter 4: Common Mistakes, Debugging, and Error Handling

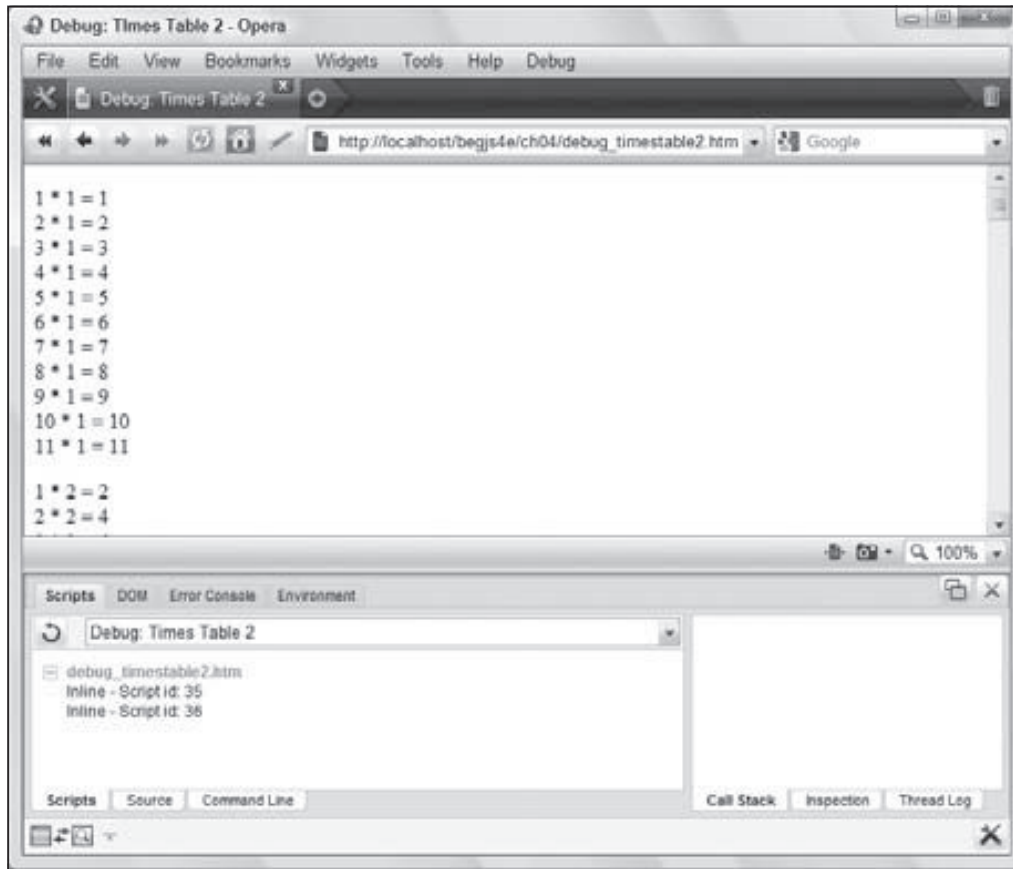


Figure 4-31

Dragonfly follows the popular two-panel layout. The left panel contains a few tabs at the top that control the several different tools offered by Dragonfly. The debugging tools are located under the Scripts tab, and it's the default tab when you open Dragonfly.

At the bottom of the left panel are more tabs: Scripts, Source, and Command Line. The Source and Command Line tabs provide the same functionality found in the other debuggers:

- ❑ **Scripts:** Displays the HTML files currently loaded into the browser and the JavaScript code they contain or reference. Figure 4-31 shows two inline scripts, meaning they are not contained in external files.
- ❑ **Source:** Displays source code. Unlike other tools, it displays only the code you select from the Script tab.

Go to the Scripts tab and click the first inline script. Doing so opens the Source tab and displays the contents of the `<script />` element in the HTML file's head (see Figure 4-32).

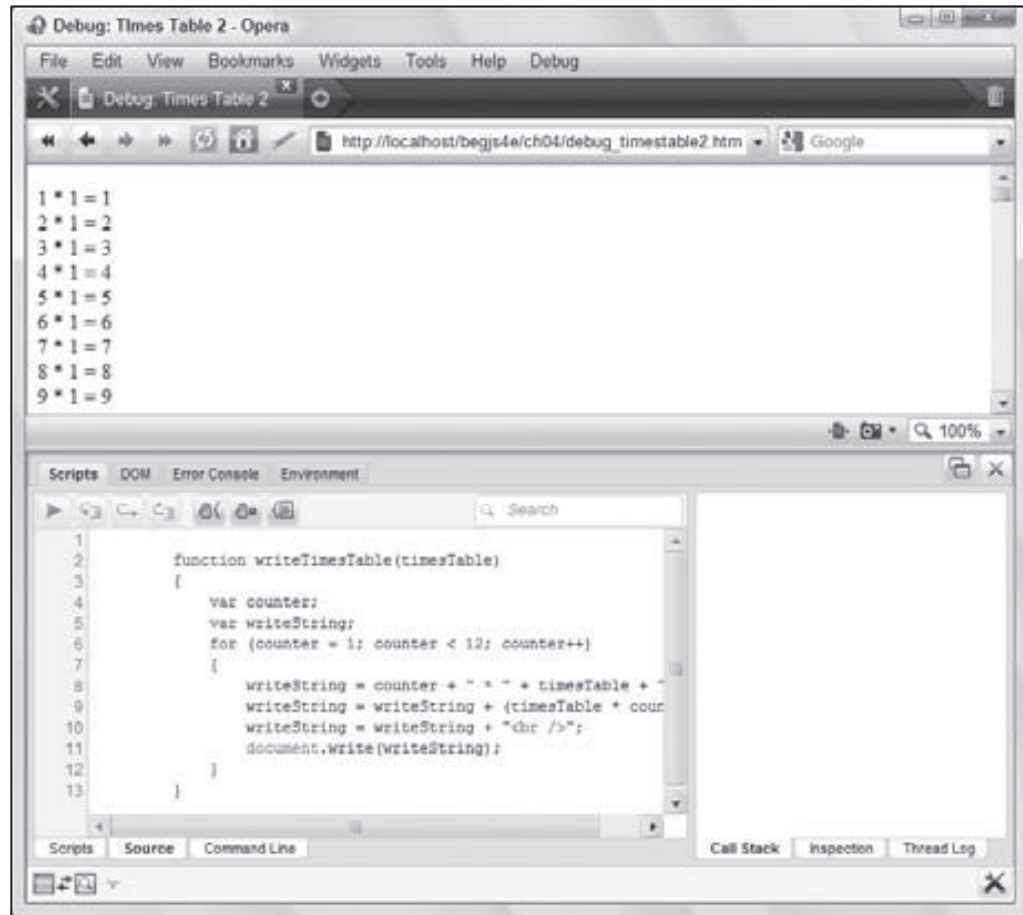


Figure 4-32

Go back to the Scripts tab (at the bottom of the window), and you'll see a pair of square brackets (`[]`) surrounding the inline script you clicked. This denotes the script that is currently displayed in the Source tab. Click the second inline script to view the contents of the `<script />` element in the HTML document's body.

- ❑ **Command Line:** This is Dragonfly's console. Like the consoles of Firebug, IE8's Developer Tools, and Safari's Web Inspector, this console allows you to check variables' values and execute JavaScript commands within the context of the currently paused line.

The right panel contains the Call Stack, Inspection, and Thread Log tabs:

- ❑ **Call Stack:** Displays the call stack.
- ❑ **Inspection:** Displays the variables and their values in scope at the paused line.
- ❑ **Thread Log:** This is an advanced debugging tool that provides the information on the processing and execution of JavaScript code. This section will not cover the Thread Log tab.

Chapter 4: Common Mistakes, Debugging, and Error Handling

Setting Breakpoints

Setting breakpoints in Dragonfly is as straightforward as in the other tools we've discussed thus far. Click one of the inline scripts in the Scripts tab to load it into the Source panel, and click to the left of the line numbers. A little black dot appears on the line number, indicating that a breakpoint is set for that line. Figure 4-33 shows a breakpoint set on line 11 of the first inline script.

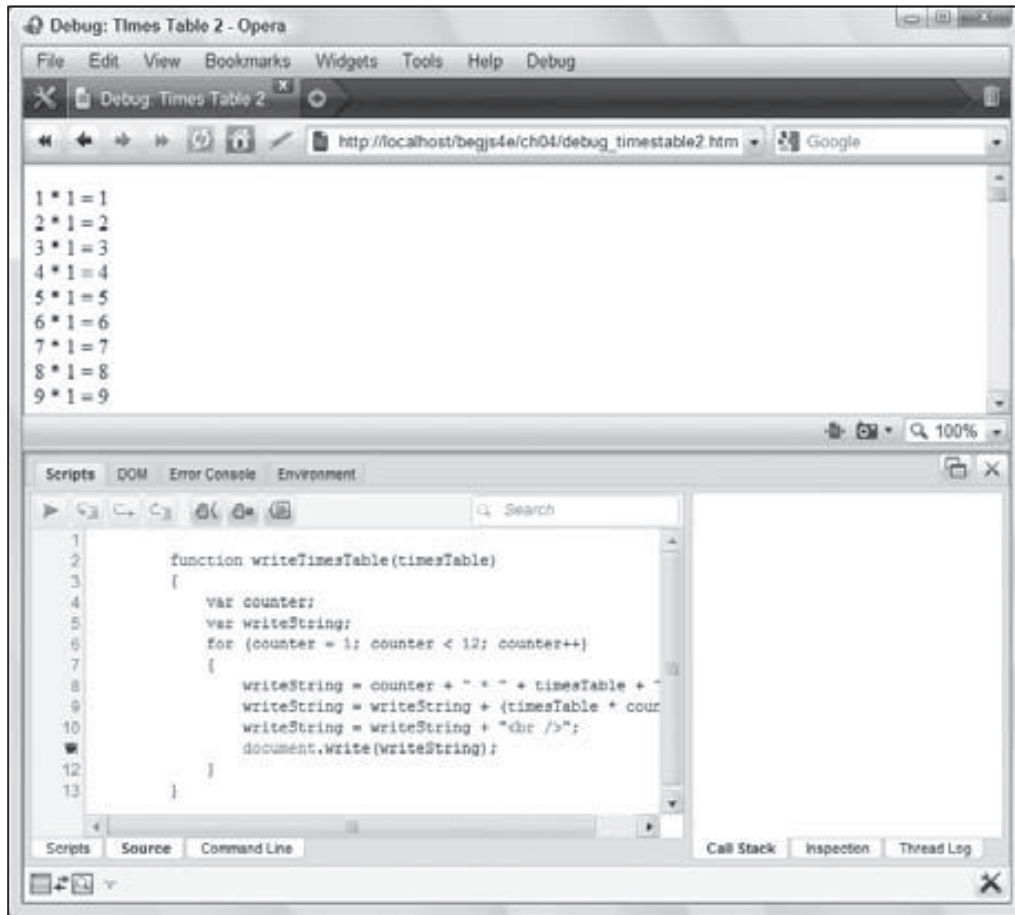


Figure 4-33

Stepping Through Code

Code stepping is controlled by a row of icons above the source code (see Figure 4-34).

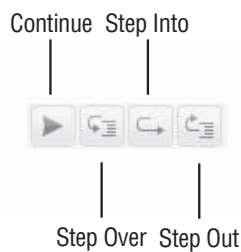


Figure 4-34

Chapter 4: Common Mistakes, Debugging, and Error Handling

These icons perform the same functions as in Firebug, IE's Developer Tools, and Web Inspector. They are, in order:

- ❑ Continue (F8)
- ❑ Step Into (F11)
- ❑ Step Over (F10)
- ❑ Step Out (Shift + F11).

Like Firefox and Firebug as well as Safari and Web Inspector, Opera and Dragonfly update the page as you step through code, allowing you to see the results of each line of code as it executes.

Summary

In this chapter you looked at the less exciting part of coding, namely bugs. In an ideal world you'd get things right the first time, every time, but in reality any code more than a few lines long is likely to suffer from bugs.

- ❑ You first looked at some of the more common errors, those made not just by JavaScript beginners, but also by experts with lots of experience.
- ❑ Some errors are not necessarily bugs in your code, but in fact exceptions to the normal circumstances that cause your code to fail. (For example, a Java applet might fail because a user is behind a firewall.) You saw that the `try...catch` statements are good for dealing with this sort of error, and that you can use the `catch` clause with the `throw` statement to deal with likely errors, such as those caused by user input. Finally, you saw that if you want a block of code to execute regardless of any error, you can use the `finally` clause.
- ❑ You looked at Firebug for Firefox, IE8's Developer Tools, the Web Inspector for Safari, and Opera's Dragonfly. With these tools you can analyze code as it's being run, which enables you to see its flow step by step, and to check variables and conditions. Although these debuggers have different interfaces, their principles and feature sets are pretty much the same.

Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

1. The example `debug_timestable2.htm` has a deliberate bug. For each times table it creates only multipliers with values from 1 to 11.

Use the script debugger to work out why this is happening, and then correct the bug.

2. The following code contains a number of common errors. See if you can spot them:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
```

Chapter 4: Common Mistakes, Debugging, and Error Handling

```
<title>Chapter 4, Question 2</title>
</head>
<body>
<script type="text/javascript">
function checkForm(theForm)
{
    var formValid = true;
    var elementCount = 0;
    while(elementCount <= theForm.length)
    {
        if (theForm.elements[elementCount].type == "text")
        {
            if (theForm.elements[elementCount].value() == "")
                alert("Please complete all form elements")
            theForm.elements[elementCount].focus;
            formValid = false;
            break;
        }
    }
    return formValid;
}
</script>
<form name="form1" onsubmit="return checkForm(document.form1)" action="">
    <input type="text" id="text1" name="text1" />
    <br />
    CheckBox 1<input type="checkbox" id="checkbox2" name="checkbox2" />
    <br />
    CheckBox 1<input type="checkbox" id="checkbox1" name="checkbox1" />
    <br />
    <input type="text" id="text2" name="text2" />
    <p>
        <input type="submit" value="Submit" id="submit1" name="submit1" />
    </p>
</form>
</body>
</html>
```